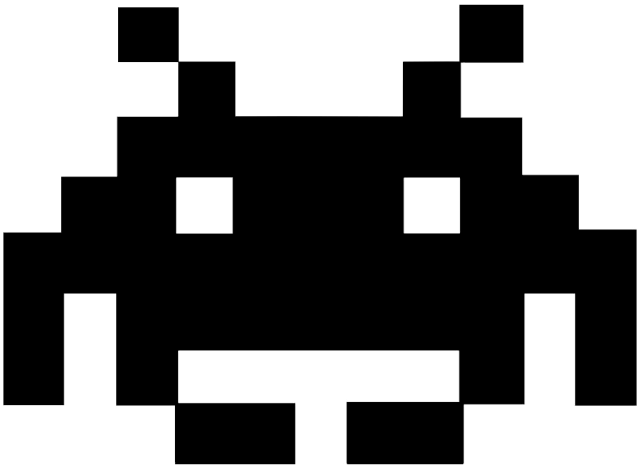
Rapport Projet POO

Space Invader - Piste Noire

Ellmo Rivetti

ESIEE
PARIS



Ellmo Rivetti
09/12/2018

E3FI

Sommaire

I. Présentation du projet

Le but de ce projet était de reproduire le célèbre jeu Space Invader. Il s'agit d'un shoot'em up où le principe est de détruire des vagues d'alien grâce à un vaisseau pouvant leur tirer des missiles dessus tout en se déplaçant horizontalement sur l'écran.

Nous devions réaliser ce programme en C# tout en utilisant les principes de programmation orientée objet. Pour ce faire, trois pistes différentes étaient proposées: piste verte, bleue et noir. La piste verte consistait offrait une architecture déjà faite, la bleue laissait le choix de l'architecture au développeur et la noire forçait à utiliser le modèle Entité-Composant-Système. Ayant déjà fait de la Programmation Orientée Objet lors de mes années d'études précédentes, j'ai décidé de faire la piste noire. Lors de la conception du projet, mon camarade Stanislaw Pluszkiewicz et moi même nous sommes entre aidé pour pouvoir bien comprendre le concept d'Entité-Composant-Système.

J'ai utilisé GitHub pour faire mon projet, voici le lien du dépot:
https://github.com/EllmoRivetti/SpaceInvader

II. Description de la structure du programme

1. Initialisation du programme

Lors de la création du logiciel, j'ai utilisé le modèle Entité-Composant-Système pour structurer mon code. La classe *Program* va créer une fenêtre Windows Form grâce à la classe *RenderForm*. A ce moment, tous les éléments graphiques se mette en place et l'instance d'un objet *Graphics* est créé. Cet objet nous permettra par la suite de dessiner les images qui seront affichées dans le jeu. Une fois l'instantiation de la fenêtre, l'objet *RenderForm* va créer un objet *Engine*. Lors de la création de cet objet, un singleton sera créé pour pouvoir accéder de manière statique à cet objet dès qu'on le souhaite.

2. L'Engine

L'*Engine* est la classe principale du programme. C'est elle qui stock toutes les entités, nodes et systèmes du programme. La liste principale est celle des entités. En effet, si une entité est créée, il faut que celle-ci soit ajoutée à la liste "EntitiesList" pour qu'elle puisse être intégrée au jeu.
La classe Engine stocke aussi trois listes de systèmes. Une liste pour les systèmes d'interaction avec l'utilisateur, une liste pour les système de fin de jeu et une liste pour tous les systèmes de traitement des données du jeu. La séparation des différents systèmes était nécessaire pour le bon fonctionnement de la fonction "Update()". Cette fonction est la boule principale du jeu. C'est elle qui va itérer sur les systèmes des listes citées précédemment pour que ceux-ci traitent les données. Le choix de la liste sur laquelle la fonction "Update()" va bouclée est définie grâce à des booléens permettant de connaître l'état du jeu :

- IsPaused est vrai : pour savoir si le jeu est en pause. Dans ce cas, la fonction itérera sur la liste ‘UISystems’.
- IsVictory ou IsDefeat est vrai: pour savoir si la partie est finie. Dans ce cas, la fonction itérera sur la liste ‘EndGameSystems’.

- Si aucun des booléens précédents est vrai: Dans ce cas, la fonction itérera sur la liste ‘SystemsList’.

La classe Engine est aussi celle qui va instancier le jeu à l’état zéro. C’est à dire qu’elle va créer les entités nécessaire au début du jeu. Une fois que la partie est terminée, il est donc facile de redémarrer le jeu grâce à ces méthodes d’initialisation.

Pour finir, la classe Engine permet aussi d’afficher les sprites à l’écran grâce à la méthode “Render(Graphics)” qui va récupérer l’instance de l’objet *Graphics* pour ensuite afficher les images. Chaque entité possède une fonction permettant d’afficher sa sprite. Lors de l'exécution de la fonction “Render(Graphics)”, une boucle s’effectue sur la liste d’entités pour afficher les sprites de chacunes. Cette fonction permet aussi d’afficher les écrans de pause, victoire et défaite grâce au booléens cités précédemment.

3. Les entitées

Les entitées représentent tous les éléments qui vont être affichés sur l’écran. Une entitée se compose composants qui permettent de stocker les données qui seront traitées durant le jeu.
Chaque classe d’entité est fille de la classe mère *Entity*. Cette classe crée l’élément principal des entitées: une liste contenant des composants. Les composants qui seront contenus dans la liste varient selon le type de l’entité. En effet, chaque entité a des propriétés différentes. Certaines doivent être affichées à l’écran, certaines sont sujettes à des collisions ou bien peuvent se déplacer. Les trois cas précédents sont des cas récurrents pour de nombreuses entités, il existe donc trois classe permettant de créer les composants nécessaires et empêchant donc de ranger les entitées dans différentes cases. La classe *Renderable* est fille de la classe *Entity*. Si une classe d’entité l’étend, cela veut dire que l’entité sera affichée à l’écran. La classe *Kinematic* permet de créer les composant nécessaires aux déplacements des entitées puis enfin la classe *Collidable* permet de créer les composants nécessaires pour la gestion des collisions.
Comme dit précédemment, chaque élément du jeu est représenté comme une entité.
L'intégralité des classes d’entités peut être trouvé dans les annexes dans le diagramme de classe des entités.

4. Les composants

Les composants sont les données des entitées. Comme dit précédemment, chaque entité possède une liste de composant qui lui est propre selon ses caractéristiques. Chaque

composant pouvant être instancié dans n'importe quelle entité, cela permet d'empêcher la redondance de code en stockant les données dans un seul objet qui sera propre à l'entité. Par exemple, le joueur (classe *Player*)et les ennemis (classe *Enemy*) ont chacun une position. Ils ont donc chacun un composant *PositionComponent* permettant de stocker les données sur leurs position dans le jeu. Grâce à cela, nous ne sommes pas obligé de créer dans chaque entité un objet *Vecteur2D* qui va stocker les informations et ou nous pourrions parfois être obligé d'écrire les même méthodes dans les deux classes.
Chaque classe de composant hérite de la classe mère *Component* cette classe crée l'élément principale de chaque composant, une référence vers l'entité à laquelle il appartient. Il existe cinq composants différents qui gère chacun un type de donnée particulier: les positions (*PositionComponent*), les hitboxs (*HitBoxComponent*), la vélocité (*VelocityComponent*), les images (*RenderComponent*) ou les données sur les tirs (*ShootComponent*) des entitées.

5. Les nodes et systèmes

Lorsqu'une entité est créée dans le jeu, elle est est ajoutée à l'*Engine* via la méthode "AddEntity(Entity)". Celle-ci va ajouter l'entité à la liste correspondante mais elle va aussi créer les nodes qui permettront alors aux systèmes de modifier les données des composants de cette entité.
Les nodes sont générales, c'est à dire que diverses entitées peuvent créer le même type de node. Par exemple, toutes les entitées qui héritent de *Collidable* feront créer une node de type *CollisionNode* qui récupérera l'instance du composant *HitBoxComponent* de l'entité . Les nodes de ce type seront alors itérées dans le système *CollisionSystème*. Pour que la création de la node se fasse selon l'entité voulue, chaque node possède une fonction "ToCreate(Entity)" permettant de savoir si la node doit être créée pour ce type d'entité.
Chaque classe de node hérite de la classe mère *Node*.

Chaque classe node fonctionne en paire avec une classe système. En effet, les systèmes modifieront les données du jeu en fonction de l'avancé de celui-ci. Comme dit précédemment, les systèmes sont appelés dans l'*Engine* au cours de la fonction "Update()" de celui-ci. Chaque classe de système possède elle aussi une fonction "Update()". C'est pour cela que chaque classe de système implémente l'interface *ISystem* qui définit la fonction "Update()". Ces fonctions ont toujours le même principe de fonctionnement. Elles effectuent une boucle sur la liste des nodes qui est propre à chaque type de système. Par exemple, lors de la fonction "Update()" de la classe *MovePlayerSystem*, la liste de node est composée de *MovePlayerNode* et l'itération se fera donc sur ces nodes. Lors de la boucle, les données des composants contenus dans les nodes sont récupérées puis traitées via diverses méthodes.

III. Problèmes soulevés lors des tests

A. Équilibrage du jeu

Tout au long du développement, il a fallu équilibrer le jeu de manière à ce que celui-ci ne soit ni trop dur ni trop facile.
Personnellement, je préfère les jeux avec du challenge, j'ai donc décidé de rendre ce space invader plus difficile mais toujours faisable. Pour faire cela, j'ai rendu la cadence de tir des ennemis plus élevée que normalement. Pour que cela ne soit pas trop compliqué étant donné que le joueur ne peut tirer qu'un seul missile à la fois, les ennemis ne se déplacent pas très vite et leur accélération au fil de la partie est légère ce qui laisse le temps au joueur de tous les éliminer.
Les bunkers ont aussi étaient sujet à plusieurs ajustements. Au début, ceux-ci été trop résistant il a donc fallu mettre en place un système de "pixels à détruire" dans les entités missiles. Lors de la vérification des collisions, le système compte le nombre de pixels du bunker qui ont été touché par le missile puis décrémente le nombre de "pixels à détruire" restant et change les pixels touchés en blanc. Une fois que le nombre de "pixels à détruire" du missile atteint zéro, celui-ci est détruit.

B. Bugs

Lors du développement, le bug majeur qui est apparu est le fait que les sprites des éléments du jeu "sortent" à droite de l'écran.

Figure 1 : Bug de la sprite coupée

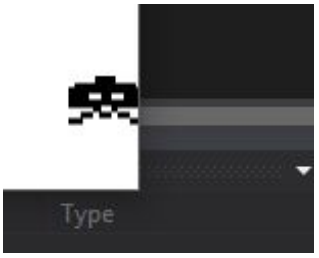


Figure 2 : Résultat attendu

On peut voir sur la première image que la sprite du joueur est coupée à droite. Le résultat que nous devrions obtenir est celui de l'image 2.
J'avais alors résolu ce bug en diminuant manuellement la distance maximum de déplacement des ennemis et du joueur ce qui fonctionnait bien sur les ordinateurs de l'école. Or, chez moi, je me suis rendu compte qu'un espace blanc inatteignable s'était formé à droite. Je n'ai pas trouvé la raison de ce bug. Selon les ordinateurs, l'espace blanc apparaît ou non.

Figure 3 : Bug de l'espace blanc

Annexes:

Diagrammes de classes ici