Rapport: Space Invaders

Adrien PELFRESNE - Alexis VAPAILLE

I - Problem Description

We need to recreate a slightly modified version of the famous game “Space invaders” with the programming langage C#. the game is played on a window’s form. The player, embodying a spaceship can shoot with the spacebar. Theses missiles need to hit enemies to eliminate them. The player win when there is no enemies left. The player enemies can also fire missiles. The player need to avoid enemies missiles in order to not die. to help him in his task, bunker are present in the map, and the player can hide under them. bunker are degrades as time goes by, when they receive missiles. You can put the game in pause when you press P key and press again on this key to resume the game.

II - Program Structure

ECS Paradigm presentation

We chose to follow the **black road**, that imply to implement our game using the paradigm ECS: **E**ntity **C**omponent **S**ystem. This paradigm change the way of seeing thing. when before, the most natural way of doing things was object oriented programming, we will now give up inheritance and reorganize our code using three main component.

Entity

An **Entity** is something that live in our scene, that can be the player, an enemy, but also a bunker or a missile. contrary to a more classical approach, we don’t need to create one class per entity. We only have one generic entity class, we gonna further attach component to this entity to model it as we want.

Component

A **Component** is a data structure, a component can be attached the an entity. a component hold informations concerning a fonction. for exemple, the Position component contains a Vec2 depicting the entity position.

System

A **System**, is the behavior part of a functionality, who will act on an entity components to implement the wanted behavior. For exemple, the ***TransformSystem*** will act on all entities

possessing the following components: “position”, “speed”, “direction” and “hitbox” to make the entity move on the scene

Entity manager

An **Entity manager** takes charge of all entities and is responsible to add or remove entities if needed. It also allows to know if an entity still exists.

System manager

A system manager takes charge of all systems and is responsible for adding them and updating the entity arrays with requirements on each loop.

Code Structure / Implementation details

Entity

The Entity class use a dictionary to store components, it make our life easier when we need to retrieve component from an entity instance. instead of getting that component by reference, we can get it by name.

```
Dictionary<String, Object> entityComponents;
```

this dictionary have Object as key to keep it generic. because having a specific Component class will not work. we can use cast to retrieve a component by name like so :

```
VelocityComponent velocityComponent = (VelocityComponent) entity.GetComponent("VelocityComponent");
```

Component

Each component has its own class, which represent a data structure, that is a class containing private field only, a constructor and properties to access those private fields.

System

Each system has its own separate class, which inherits from the abstract class ***SuperSystem***. a system must override the “Update” method. update is called every game loop and must handle component change.

The ***SuperSystem*** abstract class has the protected field “***entitesWithRequiredComponent***”. this allow overriding system to access this field, but with their specific required components. Indeed not all system require the same components from the entities. for exemple, with this behavior, ***TransformSystem*** will not be applied to an entity without position, because that entity will not be in the overridden field “***entitesWithRequiredComponent***” of ***TransformSystem***.

Coordinator

The ***Coordinator*** allow our three main components to work together. it is also our main class to handle entity creation and component attachment. Another main feature of the ***Coordinator*** is to render the scene. This is a choice we make, we could implement this in a system but, we decided to include in systems only the behavior that act inside the scene. a render function doesn't act inside the scene but render it, so we included it in the coordinator instead of having it's own system.

III - Analysis of problems/bugs solved by tests

Shooting was always going up and missile position on top of the entity sprite

when we first implemented **shootingSystem** the system that make entity fire missile, the missile position was not right in the x axis, because the missile was fired from the top left corner of the player. A quick fix was to add Player Hitbox Width / 2 to the x missile position.

another, more annoying bugs occurred when we first added enemies. if an enemy fired, the spawned missile position was at the top of the enemy, just like it was on the upper lever of the player sprite. but we needed to change this buggy behavior. we solved it by adding to our player and to our enemies a "**FacingComponent**" which tell if the entity is facing **downward** (1) or **upward** (-1). we then just needed to match the "***FacingComponent***" to adapt the missile position.

New entities were not added in arrays

At each turn of the loop, each entity is added to its associated systems so that these components are updated. When we first developed our "***UpdateEntitiesWithRequirement"*** function, we would exit the function when a requested component was not present. This prevented us from adding the entities located after. At first, we had a hard time to find the error because for us, the developed function was good. So we changed the method by replacing the return by a boolean passing to false if the component is not present in the entity.

Leaderboard file was not created and would make the game crash

When we implemented the leaderboard, everything was working fine on my machine, I pushed my code and went to sleep. when my teammate launched the game, loosing or wining cause the game to crash, it was not finding leaderboard.txt in our ressources, which is normal because the file was not created. To solve this problem, we checked if the file was present or absent in the path, if it wasn't there, we created it.

Game was broken for a unknown reason

The most annoying bug that we have to solve caused the game entities velocity to be broken. At each game loop, we update the SuperSystem's "***entitesWithRequiredComponent***" field. before, this field was a list, so each game loop we added the same entities over and over again. The fix was to switch that container to a HashSet, which make more senes because in the scene, we can't have two reference to the same entity. Entities must be unique in the scene.

IV - Conclusion

Our game works well with the ECS paradigm and has allowed us to learn a new and interesting way of coding. The beginning was a bit complicated but once we understood the system, the rest of the project went very well. In addition, we added some addons like sound, leaderboard and character animation.