Unity Programming Theory Project Plan

The idea:

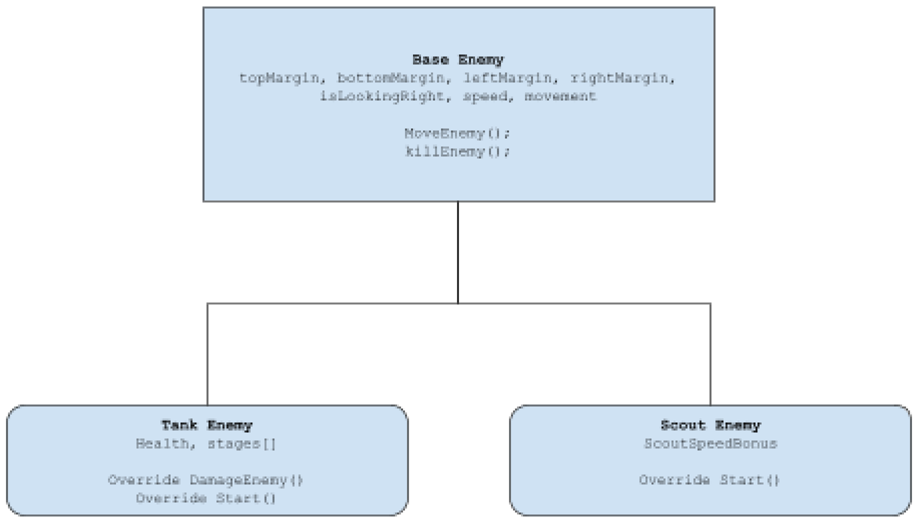
Space invaders -like game.
You control a battleship at the bottom of the screen.
You are able to shoot.
Enemies come slowly towards the bottom of the screen.
If they reach you, you lose.
You can kill them by hitting them shooting your lasers.

The OOP pillars:

- Abstraction: Will be applied in every code, no need to create anything specific about that, every good code makes use of it (and I will be sure to do so)
- Encapsulation: Like abstraction, will be used. Will mainly be manifested in interactions between player, game manager and enemies
- Inheritance: Will have different enemies which inherit from a standard enemy class.
- Polymorphism: Will be applied with enemies variation

Mechanics in depth

The player is able to move his ship left and right and shoot laser bolts at fixed intervals of time.
Enemies will move sideways until they reach a screen margin, there they will move down a bit and then start going in the opposite direction, this means they'll slowly reach the player (like the original spaceInvadersI).
Enemies Class scheme:
At the top of each you can see the name in bold, then the attributes (in the sublcasses, only the changed ones) and then the methods (again, in the subclasses only the changes).
In particular, the Base Enemy is the simplest one, its Tank variation is bigger but requires more shots to be killed, at each shot taken he will display a different colour starting from purple, then green and lastly white.
The Scout variation is much like the base one, but smaller and faster.



Pillars in depth:

Inheritance:

Both ScoutEnemy and TankEnemy will inherit from BaseEnemy class.

Polymorphism:

Every enemy will have MoveEnemy() and DamageEnemy() methods, other than the same Update() and a same base of the Start() method.

The TankEnemy will have an extra stage field, a different Start() because he needs to initialise the visual health stages and a different TakeDamage() to take all that into account.

The ScoutEnemy will have an extra field ScoutSpeed, and a different Start() to set his speed.

Encapsulation:

Interaction between player, enemies and game manager will be treated with attention, specific examples will follow:

In the GameManagerScript, we want everyone to be able to access or set the isGameOver state, but when we set the game as over, we will want to make more actions to be sure everything finishes as planned, showing a GameOver menu.

To do so, its best to use getter and setter methods.

Analogue decisions have been made regarding the enemiesLeft field, which has its own getters and setters.

The playerWon field is slightly more complicated, because its setter will have to check wether its being set to true or false and in the first case he also has to set isGameOver to true, this way setting the gamer over because the player won becomes an atomic operation and we can be sure when we reach the game ending screen we know wether its'a win or a loss.

Abstraction:

All specific code sections will have their own method, more specific examples will follow:

The update() method could get really chaotic in Unity scripts, abstraction can help solving that: in the playerScript, the logic and checks needed to make the player move have their own method playerMove(), which gets only called once in the update() method.

Analogue decisions have been made in the Update() method for BaseEnemy, having all the movement logic separated into MoveEnemy().