UNIVERSITÀ DELLA CALABRIA

Game Programming Course

SCI-FC

Game Design Document

Abstract: Development of a Sci-Fi top-down shooter through the Unity engine. The following will discuss the game dynamics and various stages of implementation, as well as the design choices of the System.

Alessandro De Simone

Index

1. Overview

1.1 Genre and Setting

SCI-FC can be defined as a top-down shooter placed in a SCI-FI context, specifically in a drifting spaceship.

We will impersonate a **Scavenger**, a person who travels in search of abandoned structures to retrieve wreckage and valuables, but this time she gets stuck inside a spaceship soon after docking.

In addition to the use of firearms, she will have the opportunity to learn certain powers (**Skills**) that will be performed by Nanites (Nanorobots) that she will accumulate during the journey, producing "magical" effects.

1.2 Game mechanics

- **Top-Down** view
- Player interactions:
    - Walking;
    - Shooting;
    - Dodge;
    - Interacting with objects in the environment;
    - Collect items;
    - Learn and use Skills;
    - Throwing grenades;
    - Activate a flashlight;
- **Goal**: Find a way out of the spaceship.

1.3 Platforms

- **PC**
  Tested on Windows and Linux
  Playable using Keyboard & Mouse, or alternatively with a Gamepad.

2. Progect Description

2.1 Project goal

The main goal of the project is to gain as much knowledge as possible about the Unity engine and the main approaches and "know-how" of Game Design. This implies interfacing with various horizontal knowledge, from Programming to Animation, in order to touch many aspects of the game engine.

2.2 External influences

Hotline Miami

Hotline Miami is a video game developed by "*Dennaton Games*" and published by "*Devolver Digital*", set in late 1980s Miami. The Protagonist explores various levels filled with enemies, often with weapons dropped at strategic points in the level or by the enemies themselves once killed. Psychedelic and with a distinctly Arcade rhythm. This project was inspired in terms of the general approach to the game, while maintaining a less frenetic style.

Synthetik

Synthetik is a video game developed and published by "*Flow Fire Games*". Unlike Hotline Miami, the camera this time is almost isometric, it is set in an unspecified future and contains typical Roguelike elements.

Setting and gunplay was inspired by this title, albeit trying to emphasize a more realistic aspect.

Both of the above titles have a fairly frenetic pace.
Therefore, to give some originality to the project, it was decided to give a slower pace to the game, making much use of the animations (sometimes absent in Synthetik, for example, to favor a more Arcade approach) and putting the Player and the enemies on the same level, who will be able to use the same weapons offered to the Player.

In addition, an Active and Passive Skill System has been implemented, helping to make the game more dynamic.

2.3 In-depth game mechanics

Protagonist

The Main Character is equipped with a certain amount of **Health** and **Shields**. While the latter regenerate over time, Health can only be restored at certain health stations, scattered throughout the levels.

In addition to moving and aiming, the player can **attack** with the basic weapon or one of the many found in the levels, accumulate and **throw grenades** of various types, consume Nanites to **use Skills**, and activate a Torch in the darkest places. The player can also dodge by rolling in the direction of movement. This last mechanic ensures the Player a certain number of frames in which he or she **cannot** be harmed.

The player will always have a basic gun, with infinite ammunition, and can carry at most one other weapon at a time, in an attempt to make the gameplay a little more strategic.

He/She can, for example, decide to keep a grenade launcher with little ammunition and use the pistol most of the time, or conversely stay safer with an assault rifle and keep the pistol only as a backup weapon.

Interactions

The player can interact with various elements in the scenario, such as:

- **Crates**: these contains weapons, randomly chosen by a weighted distribution, designed for each level;
- **Health Station**: these allow the Player to restore its health. They have a maximum number of uses;
- **Terminals**: these allow the Player to learn some Skills, randomly chosen from a pool. The player will always have two possible choices, to make the gameplay a little more varied based on its playing style;
- **Transport base:** allows the Player to access the next level;
- **Switches:** switches that trigger events of various types, for example, opening a locked door;

Collectable items

- **Ammo crates:** gives the Player an extra magazine for the currently equipped weapon. To give a little more strategy to the game, ammunition is not transferred once the weapon is changed; therefore, it is necessary to decide carefully whether to invest the ammunition crate on that weapon, rather than keeping it for a later time**;**
- **Grenades**;
- **Weapons**;

Game goal

The goal of the game, for each level, is to reach the nearest transport base in order to advance, ultimately finding a way to escape from the spaceship.

Game Over

Game Over occurs when the Health of the Protagonist drops to **zero**. In such a case, it will be necessary to start from the beginning of the level.

Levels

Each level is structured horizontally, with a series of rooms connected by corridors, with forks that can lead to advancement or finding bonuses for the Player (Terminals, ammunition, crates, etc.).

Save

Due to the nature of the game and the size of the levels, saving occurs at each Level transition.

Information such as:

- **Health**, current and maximum level
- **Shields**, current and maximum level
- Number and type of **grenades**
- Current **Weapon**
- **Skills** learned
- Remaining Skills pool
- Last achieved level

Loading occurs when the game starts, while, to avoid reading the save multiple times, passing data between levels is done through ScriptableObject, a special Unity container asset.

2.4 Interface

The interface can be divided into two macro-situations:

- Main Menu
- Pause Menu

The main menu allows you to start a new game (overwriting an existing one), continue a game you have already started, enter Settings or exit the game.

The pause menu allows you to resume the game, reload the current level, enter Settings or go to the main menu.

Both menus share the Settings section, in which you can set the level of graphical fidelity through presets, game resolution, enable or disable Full Screen, change various volumes (Music, SFX, and Global), and reassign game controls.

The game interface (HUD), on the other hand, has been deliberately kept to a bare minimum. It shows the ammunition for the equipped weapon, the currently selected Skill, the level of Nanites, the type and number of grenades selected, the current level of Health and Shields, and any messages that will appear on-screen, such as warnings or Game Tutorials.

2.5 Weapons

Although there is an (automatic) melee attack, all currently designed weapons are ranged.

Each weapon can be automatic or semi-automatic, has a certain range and an eventual degradation of damage based on distance (*e.g.*, a shotgun will make a lot of damages at very close range, while damage will drop dramatically on long distances), represented by a curve (function). It also has a reloading time, rate of fire, recoil, capacity per magazine, and a certain type of bullet.

Finally, each weapon irreversibly damages a percentage of the Nanites that enemies drop to the ground when defeated. In general, heavier weapons do more damage to Nanites.

Pistol	
Automatic	NO
Nanites damage (%)	30%
Damage	20
Rate of fire (s)	0.4
Ammo capacity	12
Reloading time (s)	1.5
Bullet type	Standard

Assault rifle	
Automatic	SI
Nanites damage (%)	50%
Damage	15
Rate of fire (s)	0.1
Ammo capacity	30
Reloading time (s)	2
Bullet type	Standard

Shotgun	
Automatic	NO
Nanites damage (%)	70%
Damage	110
Rate of fire (s)	1
Ammo capacity	6
Reloading time (s)	3
Bullet type	Shotgun

SMG	
Automatic	SI
Nanites damage (%)	60%
Damage	9
Rate of fire (s)	0.06
Ammo capacity	25
Reloading time (s)	1.8
Bullet type	Standard

Silenced Assault Rifle	
Automatic	SI
Nanites damage (%)	40%
Damage	14
Rate of fire (s)	0.1
Ammo capacity	30
Reloading time (s)	2
Bullet type	Standard

Plasma Rifle	
Automatic	SI
Nanites damage (%)	80%
Damage	20
Rate of fire (s)	0.35
Ammo capacity	12
Reloading time (s)	4
Bullet type	Plasma

Grenade launcher	
Automatic	NO
Nanites damage (%)	70%
Damage	70
Rate of fire (s)	0.7
Ammo capacity	5
Reloading time (s)	3
Bullet type	Impact grenade

Rocket launcher	
Automatic	NO
Nanites damage (%)	80%
Damage	100
Rate of fire (s)	1
Ammo capacity	4
Reloading time (s)	5
Bullet type	Rocket

Flamethrower	
Automatic	SI
Nanites damage (%)	80%
Damage	4
Rate of fire (s)	0.05
Ammo capacity	150
Reloading time (s)	2
Bullet type	--

Minigun	
Automatic	SI
Nanites damage (%)	80%
Damage	8
Rate of fire (s)	0.08-0.04
Ammo capacity	150
Reloading time (s)	5
Bullet type	Standard

Sniper	
Automatic	NO
Nanites damage (%)	70%
Damage	50
Rate of fire (s)	1.3
Ammo capacity	12
Reloading time (s)	4
Bullet type	Raycast

2.6 Bullets

There are various types of bullets in the game, to better represent the weapons available to the Player.

- **Standard**: by far the most common types of projectiles. Their movement is rectilinear and damage can degrade with distance traveled. Having a high velocity, their behavior is handled with a mix of game collisions and Raycast, a solution usually adopted when "physical" projectiles are wanted.
- **Shotgun**: share the same logic with Standard bullets, differ visually and by the type of shell casing they release
- **Plasma**: same logic as Standard, they change visually.
- **Raycast**: this type of bullet is instantaneous; it is precisely executed with a Raycast. They have a high penetrating level. The damage does not degrade with distance but after hitting a target.
- **Rocket**: Rockets are slower projectiles but emit an explosion on impact, performing area damage. Their movement is straight, and they can be hijacked with the right Skill.
- **Impact grenade**: these are grenades launched with a parabolic curve. Like rockets, they explode on impact and do area damage, albeit with a smaller range.

2.7 Grenades

Three different types of grenades are available in the game:

- **Explosive grenade**: a common grenade that explodes doing area damage, simple but effective.
- **EMP Grenade**: This grenade produces an electromagnetic pulse (EMP) that can momentarily disable enemies and their Shields (if any) (but also the Player's).
- **Slow Grenade**: This grenade adds a momentary speed malus to all entities involved in the explosion.

2.8 Explosive barrels

Scattered along the levels there are explosive barrels, of two types: classics and EMPs. Their effects are the same as those of grenades, but they are triggered once damaged beyond a certain threshold.

2.9 Skills

In the game, the Player will be able to learn various Skills, some with active elements, others passive. Some are described below as examples:

- **Force Shield**: invokes a momentary force shield that blocks all incoming bullets;
- **Tracer bullets**: makes all subsequent bullets able to track the enemy, for a given amount of time;
- **Force Pulse**: generates a shockwave that can momentarily repel and stun enemies. It can also hijack rockets;
- **X-ray**: momentarily allows the Player to see the cone of vision of enemies and their silhouette through obstacles;
- **Sprint**: momentarily increases the Player's speed;

2.10 Enemies

There are four main enemies in the game, but through management of spawn, AI level and weapons, it was possible to make the gameplay much more varied.

Basic robot

This Robot has standard Health (100) and normal reaction times. This makes it easily knocked down even with a simple gun.

They are able to listen to the sounds around them.

Advanced robot

This Robot has increased Health (150) and shorter reaction times. It is also generally **more accurate** and more ferocious, making it fearsome even on its own, with the wrong weapon.

They are able to listen to the sounds around them.

Flying drone

A flying drone equipped with a plasma rifle. It has a considerable amount of Health (200), but is slower than other enemies. It also cannot perform melee attacks, and it **explodes** few seconds after death.

He has **no** sensors to listen to sounds around him.

Small flying drone

A younger brother of the flying drone, he has a small amount of Health (40) but is faster. It too explodes after death, albeit with a smaller radius and damage.

Each enemy can be associated with a preset for AI configuration. The chosen preset is "Normal" by default, but for each game some enemies are randomly selected and the presets "***Easy***" or "***Hard***" are set for them. This distribution is chosen during level design.

Ex. *Level 'x'*

- Basic robots: out of all of them, 10% will have the "*Hard*" preset and 5% "*Easy*" preset
- Advanced robots: out of all of them, 5% will have the "*Hard*" preset
- Leave all other enemies unchanged

In addition, Robots may have randomly chosen weapons during the start of the level and may have activated Shields.

2.11 Procedurality

Various procedural situations are present in the game.

Weapon crates

Each crate randomly generates a weapon from a weighted Pool. This means that an SMG will be more likely to be generated than a Minigun.
The Pool is defined during the design of the layer.

Enemies spawn

Spawn is defined through a central location and a radius. Each of these "Spawn points" can be associated with one or more enemies to be spawned, their number, whether or not to activate Shields, and whether to spawn a random weapon as well.
By defining various Spawn points, it is possible to have semi-procedural level generation, so as to allow a good balance of game difficulty (*e.g.*, avoiding powerful enemies at the beginning of the level) and still maintain some variety between games.

IA modifier

As explained in the previous section, it is possible to define a distribution of **modifiers for enemy AI settings**, which will then be applied to random enemies in the level. In this way the Player will find, for example, basic enemies a little more hostile and advanced enemies a little below average.

3. Controls

The project was developed with mainly mouse and keyboard use in mind, but support for the Gamepad was still implemented, allowing for both devices to reassign almost all keys in the game.
Controls were handled through Unity's new *InputManager*, which primarily uses C# events, unlike the old System.

Comando	Tastiera e Mouse	Gamepad (Xbox / PS)
Movement	WASD	Left Analog
Aim	Mouse	Right Analog
Shoot	Left Mouse Button	RT / R2
Use Skill	Right Mouse Button	LT / L2
Reload	R	X / Square
Interact	E	A / Cross
Throw grenade	F	LB / L1
Change grenade	G	RB / R1
Change weapon	Q	Y / Triangle
Change Skill	Mouse Wheel	D-Pad: Right\Left
First choice	1	D-Pad: Up
Second choice	2	D-Pad: Down
Toggle flashlight	T	R3
Roll	Space	B / Circle

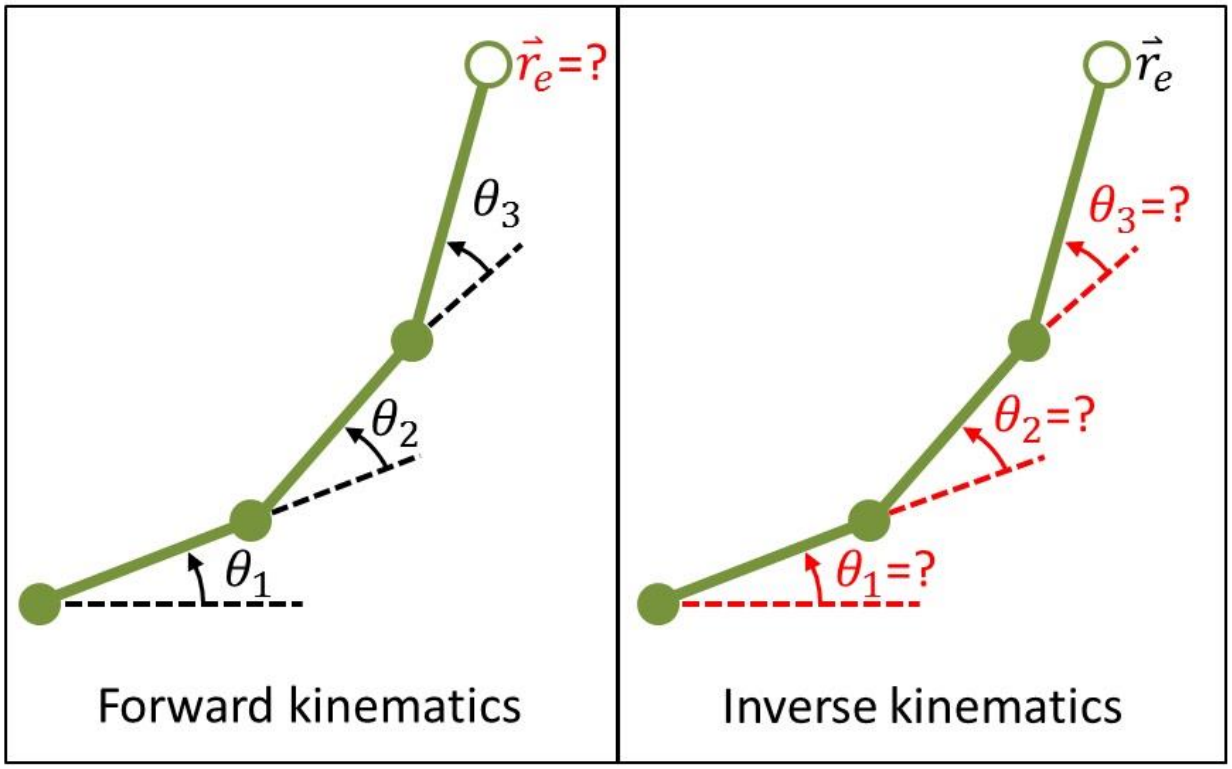
4. Assets

4.1 Animations

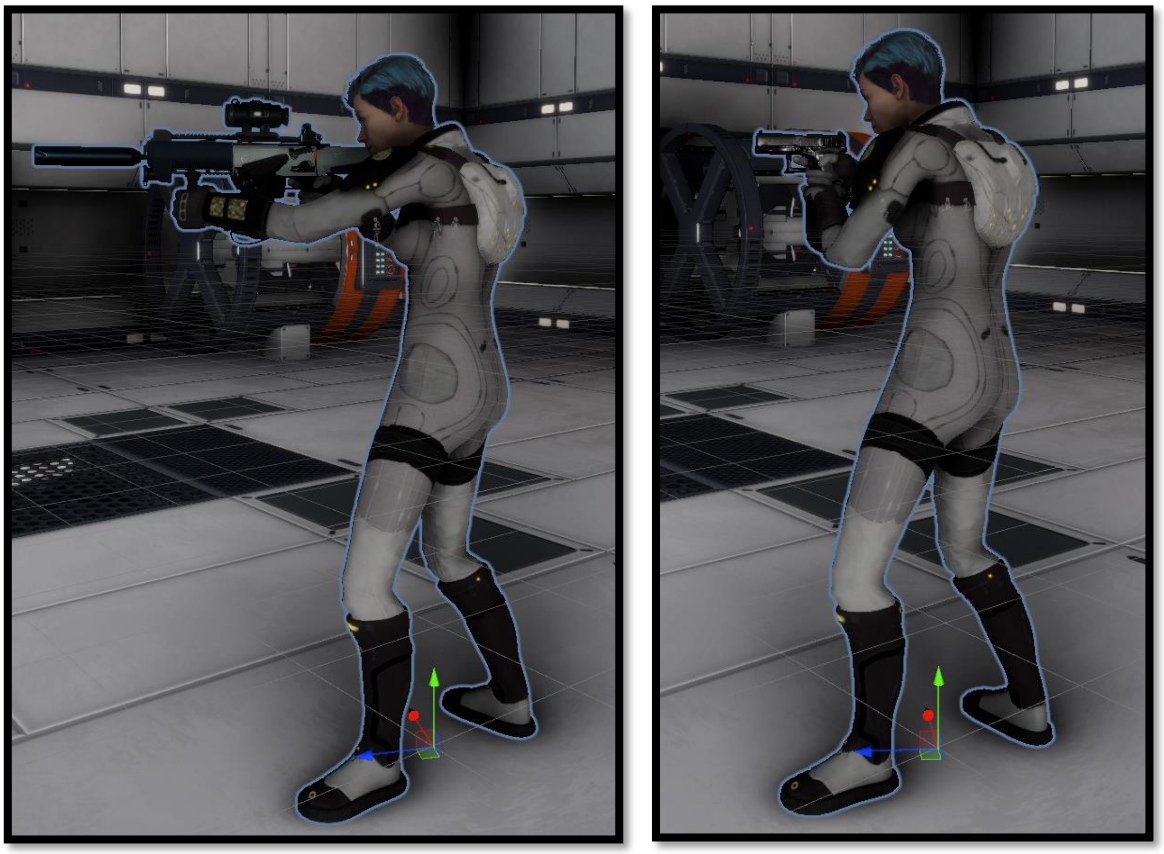
Mixamo.com, an Adobe site that allows users to map animations to their own (anthropomorphic) model, was used for almost all of the animations.

Aiming animation

In addition, to avoid defining an animation for each weapon, Unity's "**Animation Rigging**" package was used. That package provides powerful tools for model rigging, including the **Inverse Kinematics** (IK) function. While normally to make an animation we have to start by moving the outermost joint and reiterate the process to determine the position of the area we are interested in, with inverse kinematics it is the other way around. In fact, it will suffice to choose the position of the area of interest and the skeleton will be adapted accordingly.



This allowed to use a single animation for the aiming pose (idle), moving the left arm to the correct position for each weapon (an IK is applied to the left hand on the barrel of the weapon).



Animations override

When IKs could not be applied because the animation totally changed (think of the difference between reloading a pistol and a rifle, or holding a rifle and a minigun) the **Animator Override Controller** was used.

Such a Controller can be defined at design time, defining which animations to change, in order to replace the basic one at runtime.

Since such a solution would have required an Animator Override Controller for each different combination, only one was defined, for which individual animations are replaced at runtime, defined directly in the weapon that has a specific animation.

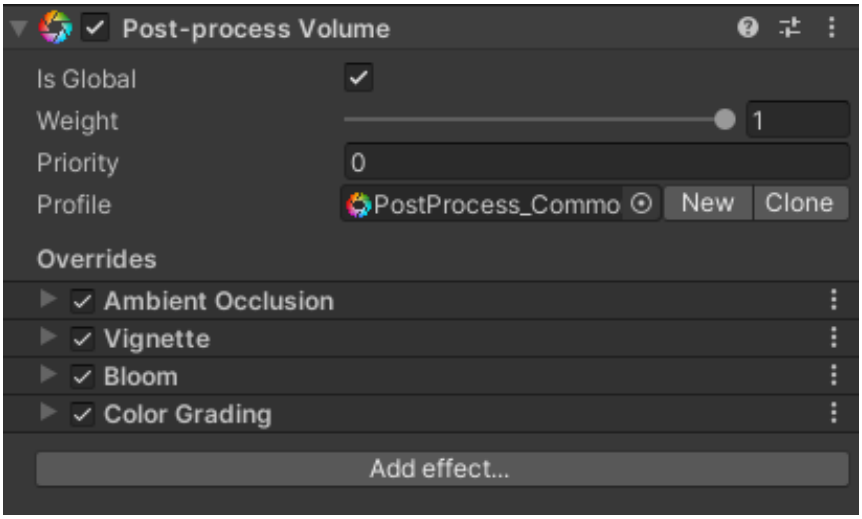
> *Es*. **Weapon reload**
> A common basic animation is used for reloading, which is overwritten when equipping a weapon with a specific reload animation, such as the Pistol or Assault Rifles.

4.2 Post-Processing

To make the levels more alive, as well as darker by giving them more atmosphere, some Post-Processing effects were applied to the game.

To apply such effects, it was sufficient to define a dedicated Post-Processing Layer, associate a "Post-Process Layer" with the camera, and add a "Post-Process Volume" to a dedicated GameObject.

In this way, effects are applied to the image buffer of the Camera before it is displayed on the screen.



- **Ambient Occlusion:** is a method of shading objects. Shadows are created near edges and between objects. Useful for giving more realistic shadow areas to the game.
- **Vignette**: is an effect in which the edges of the screen are darkened slightly, creating precisely a vignette. In this case it was used to provide feedback on the current Health to the Player.
- **Bloom**: involves simulating the effect of a very bright object by producing a kind of halo around it.
- **Color Grading**: is a simple color correction effect to the game.

5. Design and implementation

This chapter will give an overview of video game development, starting with design.

The general approach for development was a mixture of *Object-Oriented Programming* (**OOP**) and *Composition Over Inheritance*.

While an inheritance-based approach was still chosen for some aspects, for others this was replaced in favor of an approach similar to the Unity philosophy of building objects by joining together multiple Components that are responsible for performing small tasks.

In this way it was possible, for example, to implement a generic "***Explosion***" Component used for all entities that can explode. Thus, a ***grenade*** will consist of a "***Grenade" Component*** and an "*Explosion" Component*, similar to an explosive barrel consisting of "***Destroyable Barrel***" and "*Explosion.*" In addition, this approach allowed the development of variants of the same object with different behaviors, minimizing duplicate code (e.g., the various types of grenades and barrels).

Finally, to maintain a certain level of decoupling between objects, events, both classic C# (**Action**) and **UnityEvent,** have often been used, which are invoked from time to time when needed.

Some of the Design Patterns used were Strategy for weapons, Singleton for some Managers, and State for enemy AI management.

5.1 Camera

The camera is bird's-eye view, perpendicular to the X and Z axes, facing downward at a distance of 21 units from the plane.

The Component "***Position Constraint***" ensures that it always follows a binding object, in this case the Player.

In addition, the "***Camera Movement***" script offers the *CameraShake* function, which allows you to simulate camera shake, for example, when shooting.

Finally, as explained just before, a ***Post-Process Layer*** was added to that object, thus enabling the display of *Post-Processing* effects.

5.2 Giocatore

The Player entity is composed of various scripts, including:

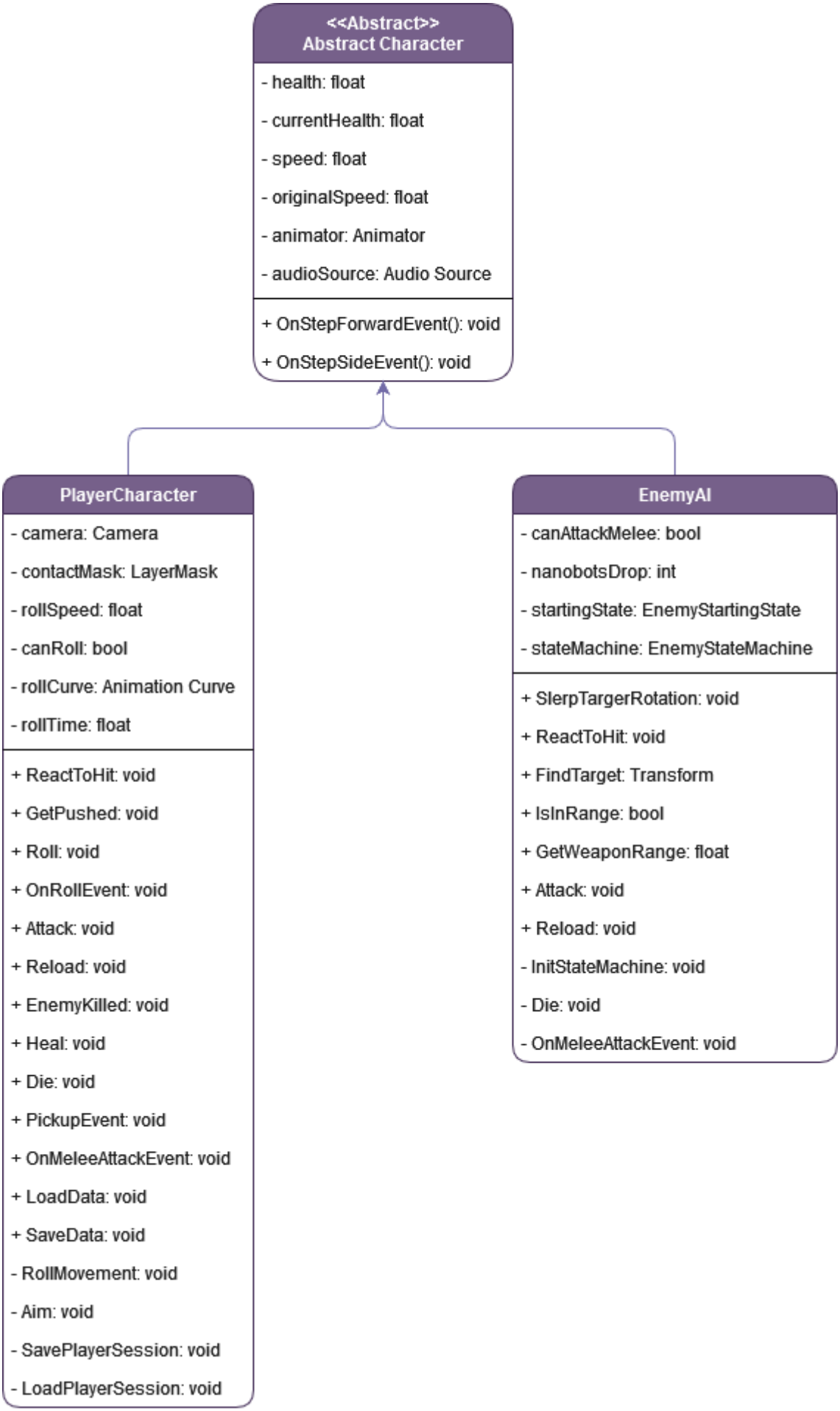
- **Player Inputs**: takes care of staying listening for all inputs, then performing the correct action for each one;
- **Player Movement**: manages the movement of the Player;
- **Player Aim**: takes care of identifying the position of the mouse (or analog) and rotating the Player accordingly;
- **Player Character**: main script of the Player. It is responsible for keeping track of current information (e.g., Health) and some of the possible actions (e.g., Attacking);
- **Weapon Holder**: script that manages access to an entity's weapons. It is responsible for keeping references to the collected weapons and configuring them once equipped (e.g., IK of the left hand);
- **Skills**: manages access to Skills. Keeps track of all collected Skills, available Nanites, collecting them, and using the currently equipped Skill;
- **Grenades Belt**: keeps track of all grenades collected, available and their launching;
- **Shield**: manages the Player's Shields, keeping track of their current level;
- **Damageable**: scripts that have all entities (animated and not) that can be damaged. Through it a C# event is invoked when the entity is hit. In the case of the Player the event is handled in "Player Character";
- **Torchlight**: manages the switching on and off of the Torchlight, as well as its consumption over time;
- **Rig Builder**: Component of the "Animation Rigging" package, handles the IK of the left hand;

- **Override Animator Runtime**: identifies the entities for which the Animator Controller can be overridden (see "Animations");
- **Ragdoll**: handles the activation of Ragdolls when the Player dies;
- **Session Player Storage**: holds a reference to the ScriptableObject used to save game data between levels. Such storage is handled in "Player Character."

5.3 Enemies

Many of the Player Components have been reused for enemies, for example the Weapon Holder, the major differences are:

- **EnemyAI**: replaces the Component Player Character;
- **Enemy State Machine**: manages the Pattern State for the AI of enemies, as well as having references to its variables;
- **Character Ears**: script that have entities that can hear external sounds, reacting to them;
- **Character Voice**: handles the activation of small lines of dialogue based on the context of the enemy;
- **GUID Generator**: script used to uniquely identify an object at runtime. In this case used to distinguish enemy type during AI configuration deployment.


<<Abstract>>
Abstract Character
- health: float
- currentHealth: float
- speed: float
- originalSpeed: float
- animator: Animator
- audioSource: Audio Source
+ OnStepForwardEvent(): void
+ OnStepSideEvent(): void
PlayerCharacter
- camera: Camera
- contactMask: LayerMask
- rollSpeed: float
- canRoll: bool
- rollCurve: Animation Curve
- rollTime: float
+ ReactToHit: void
+ GetPushed: void
+ Roll: void
+ OnRollEvent: void
+ Attack: void
+ Reload: void
+ EnemyKilled: void
+ Heal: void
+ Die: void
+ PickupEvent: void
+ OnMeleeAttackEvent: void
+ LoadData: void
+ SaveData: void
- RollMovement: void
- Aim: void
- SavePlayerSession: void
- LoadPlayerSession: void
EnemyAI
- canAttackMelee: bool
- nanobotsDrop: int
- startingState: EnemyStartingState
- stateMachine: EnemyStateMachine
+ SlerpTargerRotation: void
+ ReactToHit: void
+ FindTarget: Transform
+ IsInRange: bool
+ GetWeaponRange: float
+ Attack: void
+ Reload: void
- InitStateMachine: void
- Die: void
- OnMeleeAttackEvent: void

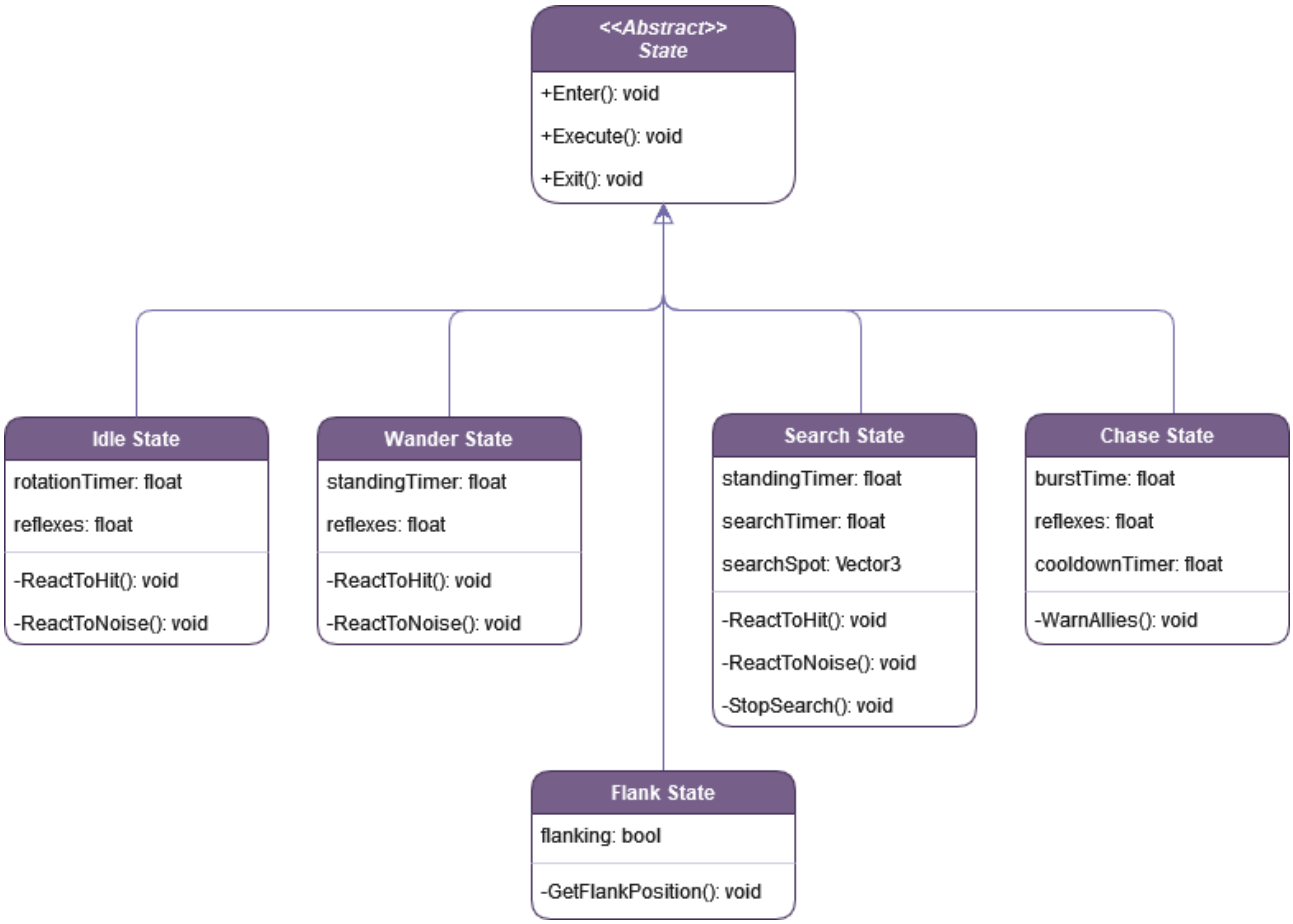
5.4 Enemies AI

As mentioned just above, enemy AI is managed through the **State** pattern. This pattern allows the behavior of an object to be changed based, precisely, on the state it is in.

Specifically, the enemy starts with an initial state, stationary ***(Idle)*** by rotating on itself or patrolling (***Wander***) from a point A to a randomly chosen point B.

From that state it is possible to go into the Search state ***(Search)*** if alerted, or into the Chase state ***(Chase)*** if the Player is identified. In the latter case, if the enemy does not have a good line of sight, it goes into the flanking state ***(Flank),*** which consists, as can be guessed, of flanking the enemy, catching him in an exposed spot.

The enemy attacks when in chasing state and the Player is in range, while backing away or physically attacking if the latter is too close.



5.5 Skills and Effects

ScriptableObjects were initially tried to be used to manage the Abilities. While functional, however, they had problems at the point when the Skill needed to communicate with other Components.

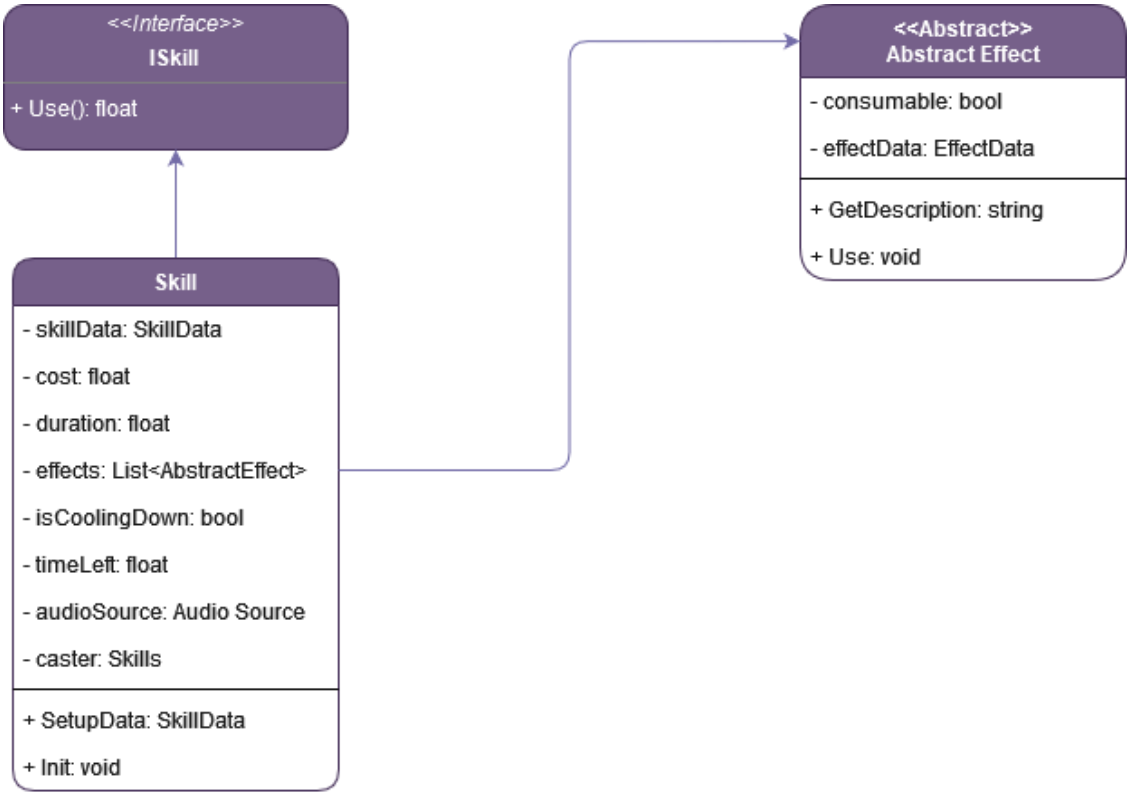
For this reason, it was chosen to implement a kind of **Composite** Pattern, adapted for use on Unity.
Each Skill, if not made exclusively of passive effects, has a certain **cost** and **duration**, as well as a list of Effects.

Each Effect derives an Abstract class that identifies whether it is consumable (passive) or not, and defines the Use() method, as well as a part that is responsible for keeping track of the metadata that will later be shown on-screen in the game.

When an Ability is used, if it is ready to be used, it invokes the Use() method for **all effects**, returning its total cost.

In this way it is possible to easily design various Skills by mixing many Effects with each other, defining, for example, simpler but cheaper Skills or more powerful but more expensive Skills.

The one responsible for keeping track of the Nanites and Skills learned is the "***Skills***" Component.



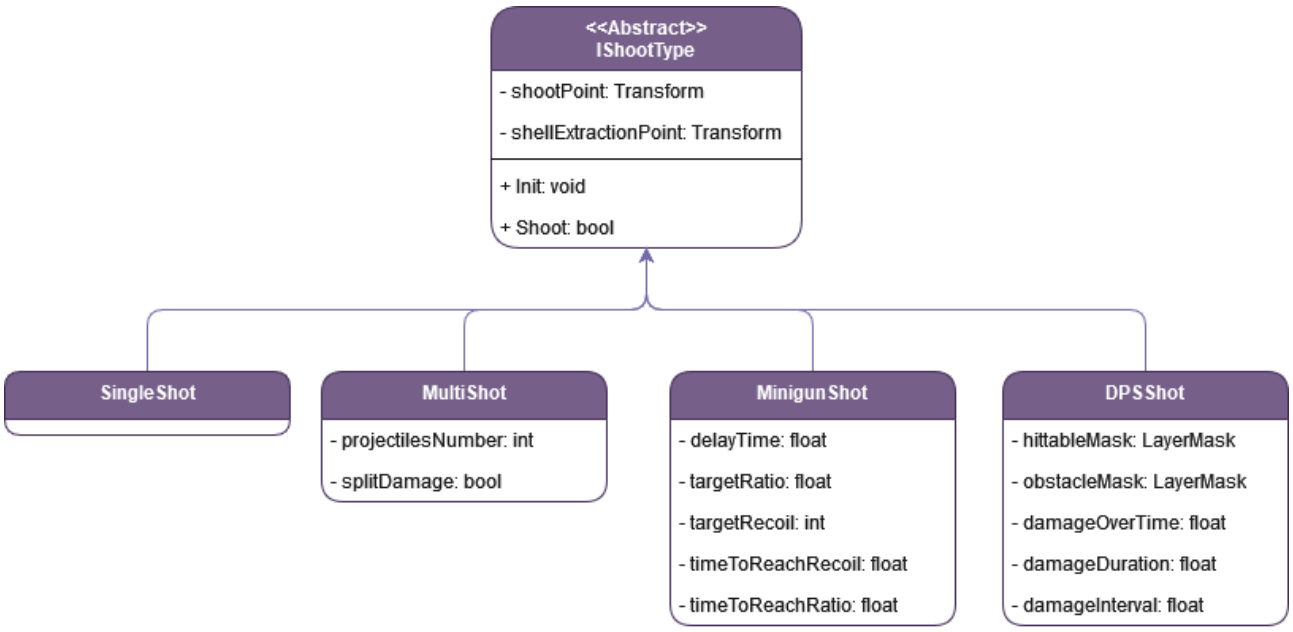
5.6 Weapons

Initially, a classical inheritance approach was chosen for weapons. Thus, there was an **AbstractWeapon class** that was inherited from the classes *Pistol*, *AssaultRifle*, *Shotgun,* etc.

Soon, however, it was realized that this approach would lead to a lot of duplicate code (between a pistol and an assault rifle potentially the firing logic remains identical). Therefore, the gun system was reengineered, using a single "**Gun**" class and applying the **Strategy** pattern to diversify the shooting method.

The Attack method of the Gun class invokes the Shoot method of the Abstract **IShootType** class, also a Weapon Component.

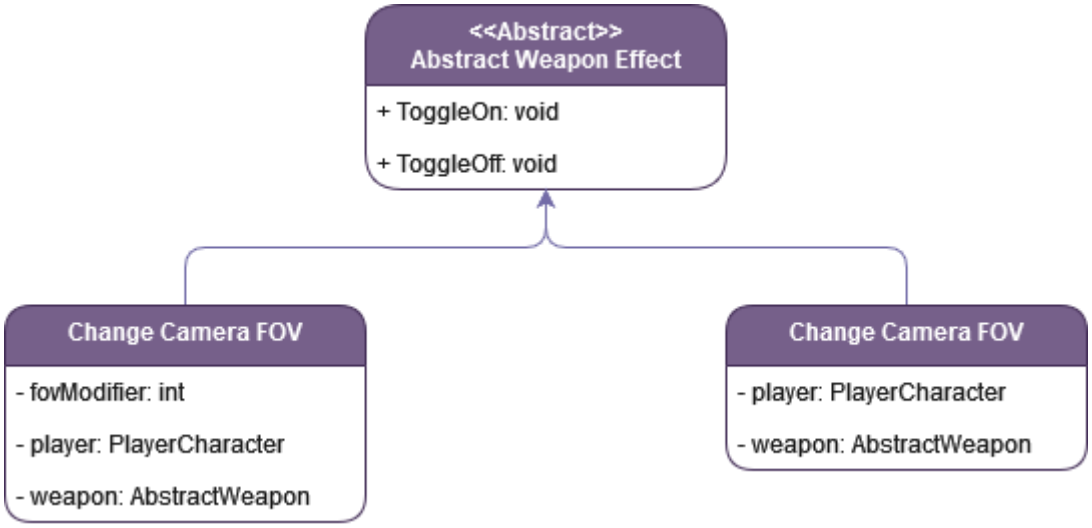
In this way, it is possible to build various weapons by changing only the *IShooType* Component in the weapon prefab. It is, for example, immediately possible to create a gun that fires several missiles per second, or to design a rifle that heals entities in a range simply by implementing a new strategy.



5.7 Weapons effects

Abilities are not the only entities that have effects. Even weapons, in fact, can apply particular variations to gameplay.

The sniper rifle slightly increases the *Field of View* (FOV) of the camera, allowing a wider area of the level to be seen, while the heavy weapons disable the dodge (*roll*).



5.8 Collection and interaction

For all objects that can be collected or interacted with, rather than defining an inheritance structure, an event-based approach was chosen.

Each object that can be picked up, in addition to the Component that defines its behavior once it is picked up, also has a "***Pickable***" Component, which is solely responsible for handling the picking logic.

Similarly, objects that can be interacted with have the "***Interactable***" Component, which handles the interaction logic (single interaction or continuous, disable interaction once the object's purpose is over, and so on).

5.9 Manager and Controller

Manager and Controller are useful objects for managing various aspects of the game, including Input, saving and loading levels.

Managers are objects that must be preserved between loading levels, while **Controllers** (with the exception of the Controller that keeps track available Skills) can be recreated at each level.

The former were then grouped into a single GameObject "**Systems**", placed inside the "***Resources***" folder so that it could be loaded at runtime, and the "***_Bootstrapper***" script was used to load that GameObject, assigning the "***DontDestroyOnLoad***" property, which precisely prevents the destruction of the object between level loads.

Manager

- **Input Manager***: holds a reference to the Component for Unity's new Input system. It also takes care of loading any key reassignments when the game starts;
- **Pool Manager***: this manager is responsible for instantiating the various Object Pools. For various GameObjects (bullets and shells), in fact, the Object Pool pattern was used, which consists of reusing the same objects instead of instantiating and destroying them repeatedly (think of the life cycle of a bullet), thus improving game performance;
- **Persistence Manager***: takes care of loading and saving game data to files, through the use of events;
- **Scene Loader Manager***: takes care of loading a scene in the game, also keeps track of the last level reached, shows the Loading Canvas, and invokes the save event between levels;

Controller

- **Coroutine Controller***: allows to start Coroutine, useful to execute for objects that are destroyed in the meantime (e.g. EMP grenades are destroyed by explosion, but the effect ends after several seconds)
- **Enemies Controller***: keeps track of enemies in the level, gives them a weapon if they need one, manages the rendering of the enemy's field of view if the X-Ray Skill is used, and gives the Player the Nanites dropped by defeated enemies;

- **Skills Pool Controller**: is responsible for keeping track of the Skills still available to be learned and for randomly choosing two skills when approaching the Terminal;
- **UI Controller**: handles everything related to the on-screen HUD, responding to Game Events. Communication between the logical part and the UI is done through the Messenger class;
- Weapons **Pool Controller**: is responsible for generating a weapon at the opening of a Crate, from a weighted pool;

*: Singleton

5.10 Saving System

Saving System is divided in two parts:

- The **Persistence Manager** is responsible for saving all game data to files. Each class that contains data to be saved implements the ***IDataPersistence*** interface and subscribes to the Manager's ***OnSave*** and ***OnLoad*** events. The interface defines two methods **SaveData()** and **LoadData()**, which pass an object of type ***GameData*** that is used to write/read data.
  When a save or load is requested, the Manager invokes the respective event, writing\reading the data to file.
  During the game, **saving** to file is called at each level transition, while **loading** from file is called at the start of the main menu and when a restart of the current level is requested.
  Some objects need to be uniquely identified upon loading (*e.g.,* Learned Skills), for this the "GUIDGenerator" script is used, which allows the generation of a GUID directly in the Inspector.

- The ***Session Player Storage*** Component takes care of storing all the data related to the Player in a *ScriptableObject, so* that it can be stored between levels. A *ScriptableObject*, in fact, keeps the values during the entire game session, while it is reset to zero after game exit.
  Saving is requested **before** a Scene change is made (using the Scene Loader Manager's "***OnLoadScene***" event), while loading occurs once the new Scene is found to be **loaded** (using Unity's Native "***SceneLoaded"*** event)

6. Future developments

There are various types of work that could be done in the near future, some of which are listed below:

- More levels;
- More Skills;
- More weapons effects; currently there are only two. Maybe give a weapon a small chance to be generated with a bonus/malus of some kind;
- Local Co-op
- More enemies, perhaps with bossfights;
- Arcade mode: endless mode in which levels are procedurally generated;
- Procedural generation of Skills, formed from set of Effects (also with Malus as *trade-offs*);
- Transposition of text (data\messages, etc.) to text files, perhaps to JSON file. ScriptableObjects are currently used, they may not be the best solution in case you want to offer multilingual selection;
- General optimization of the game;