Design Patterns Present in My Game

As one of many forms of good practice, modularity, and therefore decoupling, of code is quintessential to the ongoing work of a software developer. Being able to re-utilise elements of your portfolio again makes you a valuable developer, bringing with you the correct tools for the job at hand. My game is no different, and if I were to develop games in the future, or even allow access for another individual to see my source code, the game would be designed in such a way that elements could be reused. Design patterns also seek to create a readable, maintainable and efficient codebase, such that another person could easily get to grips with the code if need be. Below are the efforts I have made to incorporate design patterns into my work, by breaking down how and why I use each one.

Visitor

Visitors exist to decouple a repeated function from every participant of some structure, allowing a single traversing object to assert the function on each object instead. In the case of my game, projectiles act as visitors, in a much less obviously connected structure of enemies to one another. Each enemy exists as a child of the enemy controller, also acting as the FlyWeight. Any projectile sent by a player would effectively traverse this structure.

Projectiles, such as the bullets shot by both enemies and players, operate as visitors, even though they lack the exact visit/accept functions. Unity handles collision between any two objects as, essentially, visiting. When an object collides with a second object, both gain the ability to refer to each other. This allows the projectile to hold all the functions it needs to afflict anything onto any given object it collides with, and potentially multiple objects, if the bullet has the ability to pierce entities. This decouples the objects from each other, utilizing visiting projectiles to interact with any given object, so long as it has a health value. Once the purpose of the bullet is served, it can be destroyed to preserve resources.

Factory

The purpose of a factory is to separate the need to mechanism by which you instantiate a set of objects. These objects could be in large numbers, or just rather convoluted to make individually within a script. As a result, a script is written to specifically instantiate a type of object, with minimum required arguments going forward.

All prefabs are generated as children of a class that has a designated factory, spawning in objects, and then passing on all required specifics for that class to operate. A good example of a factory in my code will likely be the enemy factory, which spawns in enemies, and assigns them their Object Types. This enables easy creation of enemies from prefabs, and allows me to quickly summon enemies using debugging tools if necessary.

FlyWeight

FlyWeights follow from the idea that duplicate code is wasteful, by essentially decoupling unnecessary elements from the object they should relate to, and making each object just refer to the single instance. I can remove all the shared components and implement them within some parent spawning class, working as a means to have just a single instance of unchanging data, rather than spawn every entity with its own hard copy of identical data, in order to significantly streamline all entities spawned into the scene from prefabs.

An example within my game will likely be enemies, of which there are a few types, but they all will share common variables, such as damage values, the location of the target, and their movement speed.

Type Object

In order to specialize objects of the same type, such as enemies, or projectiles, I can use the Type Object design pattern, providing an additional script to each object of a given type, importing all the specific functions designed for that subtype of object. There are currently 3 types of enemies planned for my game, a melee chaser, a shooter, and a stationary shooter. Each of these types will be specified by their own script.

Game Loop

Game loops are one of the most critical pieces of game design patterns to have been formed. It seeks to solve the issue of the broad variety of hardware that game developers often need to accommodate for. Most hardware is designed for many purposes, and as a result, to be able to deal with the differences in processor clock-speeds, the game executes instruction on a unified ‘Tick’ generally occurring a certain number of times a second. This way, the game doesn’t actually increase/decrease in pace based on the speed of the processor running the game.

The game loop system is provided by the game engine itself, and I can tie instructions to it using the update methods available to me.

Update Method

In order to tie scripts to game ticks, I must use the update functions. There are a few available to me that allow me to control the order in which scripts are executed. The first, most prioritized function is the awake function, which happens as soon as the object is instantiated, before its initial start call. The start call then runs, frequently setting up initial script relationships with the controller, and any other required objects. The update and fixedUpdate functions are then called every tick; while update is tied to the graphical frame rate of the game, which can change due to processing limitations, fixedUpdate is always executed every tick. Every class within my game operates alongside the Game Loop, and so each class must implement at least one of these Game Loops updater.

Bibliography

Ampatzoglou, A., & Chatzigeorgiou, A. (2007). Evaluation of object-oriented design patterns in game development. *Information and Software Technology*, *49*(5), 445-454

https://www.habrador.com/tutorials/programming-patterns/

http://gameprogrammingpatterns.com/game-loop.html

Sharif, M., Zafar, A., & Muhammad, U. (2017). Design patterns and general video game level generation. *International Journal of Advanced Computer Science and Applications*, *8*(9).

Gamma, Erich. Design Patterns Elements of Reusable Object-Oriented Software. 37th printing. ed., Addison-Wesley, 2009.