Universidad Argentina de la Empresa

Diseño de Sistemas de Videojuegos

Reporte trabajo práctico

Programación de sistemas de videojuegos

Alumno

Leon, Matías

2do cuatrimestre 2024

Introducción

En este reporte voy a explicar las implementaciones de patrones hechas al trabajo práctico con el objetivo de optimizar código y corregir sus errores.

Para este trabajo se tuvo que analizar un proyecto de Unity el cual consistía de 4 objetos en la escena que instanciaban GameObjects constantemente. Dichos GameObjects, a los que me voy a referir como goblins, generaban un sonido al ser instanciados y al morir generaban un sistema de partículas y otro sonido. Estos goblins, durante su tiempo de vida, corren hacia un objetivo y explotan al llegar. Para saber qué camino recorrer, los goblins utilizan un sistema de navegación llamado NavMesh.

Este proyecto contiene cosas con las que nunca había trabajado antes como el ParticleSystem y el NavMeshNavigator así que ojala que no me generen problemas (Spoiler lo van a hacer).

Problemas encontrados

1) Los goblins se estaban creando y destruyendo constantemente. Crear y destruir GameObjects es un proceso costoso para Unity que se trata de minimizar lo máximo posible. El mismo problema ocurría con los sonidos que generaban y con las partículas.

2) Para crear un objeto, siempre se llamaba a la función Instantiate() de donde se quería crear. Esto no es un proceso ineficiente pero siempre se busca poder crear objetos a partir de sí mismos para ocasionar la menor cantidad de problemas específicos de cada objeto. Este problema ocurría con los spawners instanciando goblins y los goblins instanciando reproductores de sonido y partículas.

3) Todos los goblins guardan dentro de cada instancia información genérica que nunca es modificada. Este problema genera que se utilicen espacios de memoria innecesarios ya que todos los goblins comparten esa información pero cada uno la guarda de manera individual.

Soluciones

1) Para solucionar el primer problema decidí implementar el **patrón POOL**.

   Este patrón trata de reducir la creación y destrucción de objetos a los justos y necesarios. Lo hace generando una cola de objetos en donde se guardan todos los que no se están usando para que la siguiente vez que se quiera crear un objeto, en vez de generar otra instancia, se otorga un elemento de la cola y se remueve de la pool. Si al momento de pedirle un objeto a la pool, esta está vacía, recién en ese momento es que la pool se encarga de crear el objeto. Y en vez de que se destruyan los objetos, estos se apagan y se guardan en la pool.

   Para el proyecto traté de implementar una pool para los enemigos, una segunda para los sonidos y una tercera para las partículas. Lamentablemente no fui capaz de crear una para las partículas.

   Ahora los spawners en vez de crear goblins, le piden a la pool *EnemyPool* por un goblin, si la pool no tiene uno, esta le pide a enemy que cree un clon. Este clon se lo da la pool al spawner. Luego de que el goblin termina su ciclo de vida, este se envía a sí mismo a la pool para ser guardado.

   El funcionamiento de la pool para los audios es muy similar, el script *EnemySFX* le pide a la pool *AudioPlayerPool* por un *AudioPlayer*, si la pool no tiene uno, esta le pide a *AudioPlayer* que cree un clon. Este clon se lo da la pool al *EnemySFX*. Luego de que el *audioPlayer* termina su ciclo de vida, este se envía a sí mismo a la pool para ser guardado.

2) Para solucionar el segundo problema decidí implementar el **patrón PROTOTYPE.**

   Este patrón busca que los objetos se creen dentro de sí mismos. Esto es para evitar problemas en base a situaciones específicas de cada objeto.

   Para el proyecto traté de modificar los lugares donde se instanciaban los enemigos, los sonidos y las partículas. Implementé una función Clone() dentro del script de Enemy y de AudioPlayer, también me di cuenta que no se puede programar dentro de un ParticleSystem por lo que este patrón no lo implemente con las partículas.

   Ahora las pools de cada uno de estos objetos llaman a la función Clone() correspondiente en vez de instanciarlos ellas.

3) Para solucionar el tercer problema decidí implementar el **patrón FLYWEIGHT.**

   El concepto de este patrón es guardar información que es utilizada por muchos objetos a la vez. Estos objetos entonces dejan de guardar esta información de manera individual y en cambio tienen una referencia a la información guardada en otro lado. Esta es información que no se puede

modificar por los objetos debido a que se estaría modificando a todos los objetos que la referencian. Este patrón ayuda a ahorrar memoria RAM.

Yo note que los goblins estaban guardando individualmente información genérica que era la misma para todas las copias. Los sonidos y los efectos de partículas guardadas en RandomContainers<>. Para solucionar este problema cree un scriptable object para los sonidos de muerte, los sonidos de spawn y para las partículas. Ahora todos los goblins en vez de guardarse cada uno la información, la toman de referencia de los scriptable objects.

4) Además de estos patrones, implementé un cuarto llamado **patrón SERVICE LOCATOR**.

El service locator es un patrón que se utiliza como una lista de referencia a diferentes servicios. Es un Singleton que puede ser accedido a través de una instancia static para pasarle o pedirle algún servicio.

Para este trabajo yo utilicé el service locator para tres servicios. Para el servicio que guarda todos los town centers y devuelve alguno que esté vivo, para la pool de goblins y para la pool de AudioPlayers. Estos servicios se suscriben solos al service locator.