PROGRAMACIÓN AVANZADA EN UNITY3D

- Patrones de Optimización -

Enrique Sainz para Level-Up

Patrones de Optimización (I)

- Desde que se popularizaron los videojuegos (197X) hasta la actualidad (2022), la **potencia de los dispositivos** electrónicos ha crecido **exponencialmente**

- La mayoría de aplicaciones ya **no necesitan preocuparse** de obtener un gran **rendimiento** del ordenador
  - Incluso con un estilo de programación sencillo, y sin tener en cuenta el hardware al que nos dirigimos, la mayoría de operaciones son **casi instantáneas**

- ***No es el caso de los videojuegos***
  - Mejor rendimiento suele implicar más **espectacularidad**, más **posibilidades de interacción**, y más **ventas**

Patrones de Optimización (II)

- Hoy veremos cuatro patrones que nos ayudarán a obtener una importante ventaja en rendimiento
  - **Data Locality:** aprovecharemos nuestro conocimiento del HW para **optimizar la velocidad a la que accedemos a memoria**
  - **Dirty Flag:** nos permitirá **minimizar** el número de veces que se realizan **operaciones interdependientes**
  - **Object Pool:** nos ayudará a **reutilizar objetos**, evitando el “overhead” propio de la instanciación/destrucción
  - **Particionado Espacial:** acelera mundos virtuales compartimentando las posiciones de sus habitantes

- Data Locality -

Data Locality: El problema

- La **velocidad de proceso de nuestras CPUs** se ha incrementado **exponencialmente** año tras año en los últimos 30+ años
- La **velocidad de acceso a la memoria NO** se ha incrementado **exponencialmente** año tras año en los últimos 30+ años
- Si nuestra CPU tiene “a mano” (**en caché**) todos los datos que necesita, **aprovechará al máximo su tiempo**...
- ... pero, en caso contrario, el **acceso a memoria** se convertirá en el **cuello de botella**

Data Locality: Caché y Caché Miss (I)

- A día de hoy, l**a memoria se accede en páginas**
  - Cuando la CPU r**equiere acceder a información** (datos que procesar, código que ejecutar) obtiene **una página de la RAM** y la almacena en su caché
    - Las CPUs disponen de **diferentes niveles de Caché** (L1, L2, L3...)
    - Cada nivel es **más rápido** que el anterior
    - En nuestras explicaciones **asumiremos un único nivel** de caché
  - Si la información que requiere la CPU **se encuentra en la caché**, se conoce como **Caché Hit**
    - No hay **nada más que hacer**
  - Pero si la información **no está en la caché**, se conoce como un **Caché Miss**
    - Hay que **pedir** a la RAM que transfiera datos a la caché y **esperar** a que estos lleguen
- ***¡Las esperas por Caché Miss pueden suponer una diferencia de hasta 50 veces en tiempo de ejecución!***

Data Locality: Caché y Caché Miss (II)

- Pero las **Cachés** no cargan una única página: aprovechan para **cargar varias páginas adyacentes**
  - Si las páginas son de **código**, es alto probable que éste se **ejecute secuencialmente** a lo largo de varias páginas
  - Si son **datos**, en muchos casos esta **asunción** también es una posibilidad

- ¿Y si **colocamos** nuestro código y datos de modo que aprovechemos esta **carga adyacente**?

Data Locality: Definición

- Consiste en distribuir **los datos y el código** para aprovechar el cacheo de la CPU

- Esto implica que **poco código**...
  - (idealmente que quepa en **una sola página**)

- ... trate una **gran cantidad de datos**
  - (**contiguos y secuenciales**)

Data Locality: Implementación

- El **enfoque particular** de cada caso puede ser muy **variado**, pero hay varias consideraciones importantes
  - El **patrón Component** es perfecto para distribuir el código y los datos en **partes más pequeñas**

  - Para asegurar que los datos queden **contiguos**, lo mejor es utilizar un **array**
    - A priori, una **lista** parece más adecuada para **insertar y extraer** elementos...
    - ... pero **a la larga** sale más a cuenta:
      - **añadir** los nuevos objetos **al final**
      - **“mover” (copiar)** los objetos del final a los huecos **al borrar**
    - Existen **otras estrategias**, para casos especiales (ordenado, etc.) pero lo importante es **mantener los datos contiguos** dentro de lo posible

Data Locality: Pros

- Para un mismo proceso, podemos obtener una **mejora espectacular** (5+ veces más rápido)
  - Algo así ocurrió con el **Far Cry**, que presentó gráficos espectaculares en **consolas menos potentes** que los PCs de la época

- Programar pensando en aplicar **Data Locality** acaba resultando en **código muy desacoplado**

Data Locality: Contras

- Programar de esta manera puede resultar **poco intuitivo** para aquellos que vienen de una **POO** más **académica**
  - En cambio, para los usuarios de Unity resulta de lo más natural

- Para aprovechar la **Data Locality** al máximo tenemos que evitar acceder a **otros componentes**
  - Esto en muchos casos es **inevitable**
  - La estrategia de **intercambio de datos** se vuelve entonces crucial a la hora de no perder el rendimiento que hemos ganado con este esfuerzo
  - Estrategias como separar los datos en **hot y cold** pueden ser interesantes
    - **Hot:** caliente, se refiere a datos que **cambian muy a menudo**
    - **Cold:** fríos, serían datos que **no cambian demasiado**

- Dirty Flag -

Dirty Flag - Definición (I)

- El patrón **Dirty Flag** nos permite **retrasar tareas** hasta que sean realmente necesarias

- Imaginemos una **jerarquía de transformadas**
  - ***coche -> eje delantero -> rueda izquierda***

- Para poder dibujar la rueda, es necesario calcular la **matriz de transformación final** de ésta
  - *[Matriz Coche] * [Matriz Eje] * [Matriz Rueda Izquierda]*

Dirty Flag - Definición (II)

- Esta operación la tendremos que realizar **cada vez** que cambiemos la rueda
  - *[Matriz Coche] * [Matriz Eje] * [Matriz Rueda Izquierda]*
- Pero, ¿y si después de la rueda, también **gira el eje**? Habrá que **volver a calcular**, tanto para el eje como para la rueda
  - *[Matriz Coche] * [Matriz Eje]*
  - *[Matriz Coche] * [Matriz Eje] * [Matriz Rueda Izquierda]*
- Pero, ¿y si después del eje, **también se desplaza** el coche?
  - *[Matriz Coche]*
  - *[Matriz Coche] * [Matriz Eje]*
  - *[Matriz Coche] * [Matriz Eje] * [Matriz Rueda Izquierda]*

Dirty Flag - Definición (III)

- Esto quiere decir que, en un **caso especialmente malo**, podríamos tener que hacer **todos estos cálculos**:
  - *[Matriz Coche] * [Matriz Eje] * [Matriz Rueda Izquierda]*
  - *[Matriz Coche] * [Matriz Eje]*
  - *[Matriz Coche] * [Matriz Eje] * [Matriz Rueda Izquierda]*
  - *[Matriz Coche]*
  - *[Matriz Coche] * [Matriz Eje]*
  - *[Matriz Coche] * [Matriz Eje] * [Matriz Rueda Izquierda]*

- Lo peor de todo, es que el resultado de los cálculos marcados en rojo **ni siquiera se llegarán a utilizar**

Dirty Flag - Definición (IV)

- La solución es **marcar con un Dirty Flag** las transformadas que han cambiado

- Posteriormente, **limpiamos el flag** de aquellas que se vayan recalculando

- Dado que al **recalcular** la transformada del coche, recalcularemos **también las de sus hijos**, los flags de éstas se limpiarán, y **sólo se recalcularán una vez**

Dirty Flag - Implementación

- La implementación es muy directa, basta con **añadir un booleano** al objeto sobre el que queremos utilizar este patrón

- Son más interesantes las consideraciones de **cuándo debe utilizarse**:
  - Cuando los **datos primarios** (las transformadas) cambien más a menudo que los **datos derivados** (las matrices)

  - Cuando el **rendimiento** que ganamos justifique la **complejidad** que añadimos

Dirty Flag - Pros y Contras

- El principal pro del **Dirty Flag** es que nos **acelera casos como este** que hemos especificado

- La principal contra es que el **número de casos** que resuelve es bastante limitado
  - No es un patrón muy **reutilizable**
  - Pero para casos como el de las **transformadas y sus matrices**, es el rey

- Object Pool -

Object Pool - Definición (I)

- El patrón **Object Pool** permite mejorar el rendimiento y el uso de memoria **reutilizando objetos** de un “fondo” (pool)

- Cada vez que necesitamos un **nuevo objeto** (por ejemplo, una bala) solemos **instanciar** un nuevo prefabricado; para ello
  - Unity **pide memoria** al operativo, que tardará un poco en contestar
  - Con esa memoria, **inicializa los datos** del objeto; normalmente clona el prefabricado dato a dato
  - Cuando el objeto se deje de usar, Unity **abandona la memoria**
  - El **Garbage Collector** la reconoce como no utilizada, y la devuelve al operativo
  - El operativo la **recupera** y la marca como **reutilizable**

Object Pool - Definición (II)

- Este proceso es lo suficientemente **costoso** para que decidamos que es **interesante reutilizar** los objetos
  - Cuánto ni más si, como en el caso de un juego **“Bullet Hell”**, la cantidad de **objetos instanciados por segundo** es altísima

- Pero además, puede haber problemas de **fragmentación de memoria**
  - Muchos **chunks de memoria pequeños** que impiden que cojamos un **chunk de memoria grande**

- Para solucionar todo esto, es interesante el **Object Pool**
  - **¡OJO!** Esta solución es **interesante en C++**, donde tenemos un control de dónde se crean los objetos y cómo los referenciamos
    - En C# es bastante más complicado, sólo es posible con ***Value Types***

  - En Unity3D, lo mejor es utilizar la clase

Object Pool - Implementación (I)

- Implementaremos el **Object Pool** sobre una clase **PoolXXXX**
  - Por ejemplo, **PoolBalas**

- Dicha clase dispondrá de los siguientes **métodos**:
  - ***Init()*** - permitirá especificar la cantidad de elementos que tiene que tener el pool
  - ***Create()*** - permite obtener el primer elemento libre del pool
  - ***Release()*** - permite devolver un elemento al pool
  - ***End()*** - libera todo el pool

Object Pool - Implementación (II)

- **¿Quién resetea las variables que puedan haber cambiado?**
  - Si las resetea el que llama a ***Create()***, se puede ahorrar mucho reseteando sólo aquellos datos que puedan haber cambiado (**datos Hot**)
    - Pero esto provoca un **acoplamiento**: cualquiera que llame a ***Create()*** tiene que conocer los detalles de la clase gestionada
  - Si las resetea el Pool, es más fácil y localizado mantenerlo
    - Pero perdemos la oportunidad de **resetear únicamente** los datos que puedan haber cambiado...
    - ... incluso de **no resetear datos**, a sabiendas de que quien llama a ***Create()*** va a sobreescribir todos los datos
      - Preguntadme por **los triángulos de Commandos 2** para la **PlayStation 2**

Object Pool - Implementación (III)

- **¿Podemos hacer esto en C#?**
  - En C# no tenemos control de cuándo los objetos se instancian de manera correlativa
    - Hay que hacer un new para cada objeto, y nada asegura que queden correlativos
  - Sí que podemos hacerlo con los tipos por valor
    - Esto incluye las structs, lo cual es bueno...
    - ... pero en C# las structs están muy limitadas
- ¿Qué alternativa tenemos?
  - El ***ObjectPool<T0>***
    - Unity - Scripting API: ObjectPool<T0> (unity3d.com)

Object Pool - Ventajas

- El **coste de instanciación** se vuelve mucho más bajo
- **Adiós a la fragmentación** de memoria
- Con un poquito más de esfuerzo, estos objetos pueden estar **correlativos en memoria** y favorecer una optimización por **Data Locality**

Object Pool - Desventajas

- Sólo se puede utilizar una **cierta cantidad de objetos**, según el tamaño inicial del pool
  - Se puede crear un sistema para que vaya **creciendo de tamaño** el pool, pero **parece más óptimo** tenerlo controlado desde el principio
- Hay memoria reservada que **no se va a estar utilizando**
  - En sistemas con **poca memoria**, no parece una gran idea
  - Aunque hoy en día **hay memoria por todas partes**...
- Los objetos del **Pool** sólo pueden ser de una **clase Base**
  - Nada de derivadas en un mismo pool

- Space Partition -

Space Partition - Definición (I)

- **Nos permite localizar eficientemente objetos en un espacio particionado**

- Imaginemos un juego de rol en el que **10 jugadores** están conectados a un servidor
  - Claramente, podríamos querer **lanzar un golpe** con nuestra **Espada Butterfly Pillow Mágica +2 a acertar, +2 al daño**
    - **Notificamos** al servidor
    - El servidor **comprueba** nuestro golpe contra los otros **9 jugadores**
      - **¡Fallamos!** Normal, estamos en el desierto, rodeados por **NADIE**
      - **9 comprobaciones** tiradas a la basura

  - Ahora, imaginad esto con los otros **100.000 jugadores**... y con cada uno de ellos **golpeando una vez por segundo**
    - ¡El crecimiento es **exponencial**!

Space Partition - Definición (II)

- El patrón **Space Partition divide el espacio** en rejillas - bidimensionales o tridimensionales - y actualiza **en qué rejilla está cada elemento**
  - De ese modo podemos **descartar** fácilmente a los jugadores que están en **otra rejilla** o que no están **cerca de los bordes** más cercanos a nuestro avatar

Space Partition - Implementación (I)

- La implementación más sencilla sería con una clase Entity que únicamente pudiera desplazarse a través de
  - setters de posición
  - órdenes de movimiento

- Esta clase podría gestionar de manera estática **cuál es el cuadrante** en el que se encuentra
  - Estos cuadrantes serían estructuras que guardarían **listas de referencias** a las **Entities** que se encuentran en ellos
  - **Setear** la posición y **moverse** actualizarían los cuadrantes adecuadamente

- En el caso de Unity, este particionado se hace automáticamente
  - Las clases involucradas son Transform y Physics

Space Partition - Pros y Contras

- La **mejoría de rendimiento** es más que notable
  - Y no sólo en **juegos multiplayer**: este concepto se aplica a cualquier interacción física
- Por contra, exige que toda **comprobación** pase a través de un **mismo sistema de física**
  - Esto genera un **acoplamiento transversal**
    - Al menos en las capas de **más alto nivel** de la aplicación
  - Pero, como sabemos por **Unity y su clase Physics**, no es algo especialmente problemático
- Otra contra es que **cuando hay pocos objetos** es mucho menos eficiente
  - Digamos que **cuanto más se usa, más ahorra**
- Y otra más, es que **ocupa más espacio**
  - Lo que hoy día **no es un gran problema**