PROGRAMACIÓN AVANZADA EN UNITY3D

- Patrones de Comportamiento -

Enrique Sainz para Level-Up

Patrones de Comportamiento (I)

- Crear **videojuegos** es de por sí **complejo y costoso** en tiempo...
- ... sin tener que enfrentarse a los **problemas típicos** de **organización y compilación** de lenguajes como C++
  - Aquí viene un trozo en el que Enrique ***se tira en barril por las cataratas del recuerdo***, y os cuenta otra vez cómo pasaba prácticamente el ***50% del tiempo desarrollando motor, y el otro 50% desarrollando juego***
- Parte del problema está en **los lenguajes**
  - **C y C++**, probablemente aún los lenguajes más populares para desarrollar juegos AAA, **datan de 1972 y 1985**
  - Ambos están muy **alineados con el hardware**, y muy poco con las necesidades de los **videojuegos modernos**

Patrones de Comportamiento (II)

- Otra parte aún más importante está relacionada con la **naturaleza iterativa** del desarrollo de videojuegos
  - Diseñar — Implementar — Testear — Corregir — Rediseñar — Reimplementar...
  - Los tiempos de compilación pasan a convertirse en un problema grave de desarrollo
  - Y los diseñadores necesitan de los programadores para poner en marcha su creatividad

- Veamos **tres patrones** dedicados a facilitar el desarrollo:
  - ***Bytecode***
  - ***Type Object***
  - ***Subclass Sandbox***

- Bytecode -

Bytecode: Definición (I)

- ***El patrón Bytecode nos invita a añadir código no compilado, sino interpretado por una máquina virtual***

- Previo a motores como **Unity**, prácticamente cualquier acción que tuviera lugar en un juego estaba **programada en C/C++**
  - **Arreglar bugs** implicaba recompilar el código
  - **Añadir features** implicaba recompilar el código
  - **Permitir modding** implicaba... bueno, no se podía hacer modding sin publicar el código de nuestro juego

Bytecode: Definición (II)

- Pero... **¿existe alguna manera de codificar acciones sin necesidad de compilar?**
  - Bienvenidos al dominio de los **lenguajes interpretados**: Basic, LUA, Javascript, Perl...
- Y eso de **ejecutar código no compilado**... ¿no será un poco lento?
  - Un poco no - **un mucho**
- ¿Y no hay nada que podamos hacer? Sí: **Bytecode**
  - Básicamente podemos convertir nuestro código **en ceros y unos** que podamos interpretar como órdenes...
  - ... y preparar un intérprete de dichas órdenes que sea rápido como el rayo
  - ... o al menos lo suficientemente rápido para una serie de tareas

Bytecode: Definición (III)

- **Para qué** usamos **Bytecode**
  - Tareas de **alto nivel:**
    - Gestión del juego
    - Decisiones de IAs
    - “Cosas de diseñador”
    - Interacción eventual con el bajo nivel

- **Para qué no** usamos **Bytecode**
  - Tareas exigentes en rendimiento
  - Tareas de bajo nivel
    - Manipular el hardware
    - “Cosas de programador”

Bytecode: Implementación (I)

- Tendremos que implementar **tres piezas principales**:
  - **Editor del lenguaje -** permite editar el código
    - Puede ser un **simple editor de texto**
    - También puede ser un editor de **cajas unidas con líneas**
  - **Codificador de lenguaje -** convierte el lenguaje (o las cajas) en **Bytecode**
    - Podemos **hacerlo nosotros**
    - También podemos utilizar el de un lenguaje **ya existente** (LUA, Javascript)
  - **Intérprete del lenguaje -** interpreta el lenguaje y llama a funciones asociadas
    - Podemos **hacerlo nosotros...**
    - ... o utilizar el de un lenguaje **ya existente** a través de sus **Binds**
      - E.G. muchos lenguajes tienen **Binds** a Raylib

Bytecode: Implementación (II)

- **Interesante:** en el libro vienen **instrucciones y ejemplos muy completos** sobre cómo implementar estas piezas en C++
  - Se nota especialmente que el libro estaba dirigido a los **desarrolladores de videojuegos de aquella época** (2000 - 2010)

- En **Unity**, el **lenguaje C#** acaba siendo nuestro origen de **Bytecode**
  - **C#** se “compila” en un **Bytecode** llamado **IL (Intermediate Language)**
    - Como curiosidad, **Visual Basic** también compila en IL

Bytecode: Ventajas (I)

- Permite **que los diseñadores iteren más rápidamente**
  - Incluso permite que éstos puedan trastear ofreciéndoles un **subconjunto asequible** de código para trabajar

- **Evita tener que recompilar** todo el código a cada cambio
  - Esto **acelera** sobremanera el proceso de **iteración**
  - Realizado correctamente, permite incluso **modificar el código en caliente** (mientras el juego se está ejecutando)

- Permite **tratar el código como si fueran datos**
  - Esto implica poder hacer fixes sin tener que recompilar el código original
  - O poder descargar actualizaciones/extensiones, de nuevo sin recompilar

Bytecode: Ventajas (II)

- **Separa** el código de **alto nivel** del código de **bajo nivel**
  - Como desarrolladores, elegimos **qué funcionalidad hacemos pública** a través del **Bytecode**
  - En este sentido, es intuitivo no publicar **funcionalidad de bajo nivel** (relacionada con el hardware), sino sólo **funcionalidad de alto nivel** (relacionada con el diseño)
- Es **independiente de plataforma**
  - No necesitamos preocuparnos de **a qué ordenador** va dirigido el juego...
  - ... ni las actualizaciones, ni el contenido extra que decidamos publicar
- Permite **habilitar modding**
  - **Sin publicar el código**
  - Y si aprovechamos **lenguajes interpretados existentes**, los modders lo tendrán especialmente fácil

Bytecode: Desventajas

- **Añade más complejidad al desarrollo**
  - ¿De verdad tengo que programar un nuevo lenguaje de programación?
    - **Programador A:** *¡Genial! ¡Voy a aprender un montón! Es más - ¡desarrollaré tres lenguajes en lugar de uno! ¡Tengo tiempo infinito!*
    - **Programador B:** *¡Fatal! ¡Da igual lo caros que sean los programadores, por cada idea de cada diseñador, contrataré un nuevo programador! ¡Tengo dinero infinito!*

- **Es mucho más lento**
  - Sólo deberíamos utilizarlo si las posibilidades que aportan **compensan la pérdida de rendimiento**
    - … cosa que **a día de hoy**, con los ordenadores de hoy, viene a **compensar bastante**

- Subclass Sandbox -

Subclass Sandbox: Definición

- ***Permite definir comportamiento en una subclase, definiendo un conjunto de operaciones en su clase base***

- Imaginemos que tenemos que crear un **alto número de IAs** para nuestro juego

- Podemos **generar cada IA por separado**...
  - ... pero al cabo de poco veremos que estamos **reescribiendo gran cantidad de código**

- O podemos generar las IAs en base a un **conjunto de funciones** definido en la **clase base** de dichas IAs
  - Bienvenidos al **Subclass Sandbox**

Subclass Sandbox: Implementación (I)

- **Implementaremos una clase base**
  - IA, Hechizo, Vehículo...

- Dicha clase base **dispondra de**:
  - Métodos abstractos o virtuales a través de los que **será utilizada** desde la lógica del juego
    - ***NotifyShot()***, ***NotifyPlayerPosition()***...

  - Métodos propios que proporcionarán **funcionalidad común** a todas las derivadas
    - ***CheckVision()***, ***Move()***, ***Jump()***, ***Turn()***...

Subclass Sandbox: Implementación (II)

- **¿Qué funciones debemos proporcionar?**
  - Funciones propias del **dominio del problema** que estamos solucionando
    - El **movimiento** de la IA, comprobaciones de los **sentidos**...
  - Funciones que **no sean un simple "forward"** al sistema
    - Tiempo atrás parecía interesante **encapsular** todas las llamadas de una IA a su clase base
      - Por aquel entonces daba la sensación de que tener una función **PlaySound()** en la clase base **proporcionaba independencia** de la implementación
      - Hoy en día no parece correcto - quizá porque el código es **mucho más estable, menos cambiante**
        - ***¡Gracias a motores como Unity!***
  - En general, **todo el código que se duplique** ni que sea en dos clases derivadas debería formar parte de la clase base
    - O, como poco, de una **derivada de la clase base** que sea a su vez base de las derivadas mencionadas

Subclass Sandbox: Implementación (III)

- **¿Debemos proporcionar todas las funciones a través de la clase base?**
  - Si lo hacemos, existe el peligro de que la clase crezca mucho más allá del principio de **Single Responsibility**
  - En ese caso, **proporcionar varios interfaces** es muy interesante
    - En el ejemplo de la IA podemos utilizar las clase **Motor**, **Vista**, **Oído**, **Inventario**...

Subclass Sandbox: Pros

- Ayuda enormemente a **arquitectar el código** de un tipo de elementos (IAs, hechizos...)
- **Reduce el conocimiento necesario** de un novato (o de alguien que lleva mucho tiempo sin ver ese código) a la hora de añadir un nuevo elemento de ese tipo
- **Conduce** el proceso de creación, al punto incluso de poder **automatizarlo con Wizards**

Subclass Sandbox: Cons

- Tiene los **contras típicos** de cualquier código con **arquitectura correcta:**
  - Hay que **mantenerlo** para que no degenere
  - Necesitas **buenos programadores** para realizarlo

- De todos modos, los **Pros** suelen **ganar por goleada** a los **Cons**

- Type Object -

Type Object: Introducción

- ***Permite la creación de nuevas "clases" creando una única clase, tal que cada instancia representa un tipo diferente de objeto***

- Como ya hemos explicado, el proceso de creación de un juego tiene un **alto coste en iteraciones**
  - Diseñar — Implementar — Testear — Corregir — Rediseñar — Reimplementar...

- Si nuestro juego necesita muchos objetos con características comunes...
  - Dragones, trolls, magos oscuros... al final todos son **enemigos** y se moverán para atacarnos
  - ¡Pero por cada uno que creemos, tocará **recompilar** varias veces!

- ... o si **no sabemos de antemano** los tipos que necesitamos...
  - Por aquello del proceso iterativo

- ... entonces es un buen momento para hablar del **Type Object**

Type Object: Advertencia

- El **Type Object** tiene sentido sólo en lenguajes donde definir un nuevo tipo requiere **recompilar** todo el proyecto

- Este **no es el caso de C# en Unity** - y los addressables que ya hemos visto son buen testimonio

- Hay que entender la necesidad del **Type Object** desde la perspectiva del 2000/2010, cuando **el código en C++ dominaba la ejecución** del juego, y disponer de **Bytecode** era una fantasía

Type Object: Implementación (I)

- Hay que definir una clase **Type Object**, tal que cada instancia representa a un tipo diferente

```
class MonsterType()
{
	string name;
	float startLife;
	float startDefense;
	float startAttack;
	Mesh visuals;
	Animation[] animations;
};

MonsterType dragonType { “Dragon”, 100, 10, 10 }
MonsterType orcType { “Orc”, 10, 2, 2 }
MonsterType Shaman { “Shaman”, 5, 0, 5 }
```

Type Object: Implementación (II)

- Ahora hay que definir una **clase base** que utilizará el **Type Object** como manera de diferenciar entre tipos

```
class Monster()
{
	MonsterType type;

	float currentLife;
	int currentAnimation;

	[...]
};
```

- Sin el patrón **Type Object**, lo más normal habría sido **derivar cada tipo** de monstruo de esta clase base

Type Object: Ventajas

- Permite añadir fácilmente **nuevos tipos con poca recompilación**
  - Si el código está bien arquitectado, los cambios pueden reducirse a una librería, y el linkado es un proceso relativamente rápido

- Con un poco de esfuerzo, podemos **convertir estos tipos en datos**
  - Cargamos los tipos desde un **archivo XML** sobre un array de tipos
  - Obtenemos la referencia a los tipos desde el array **a través del nombre**
  - **¡Recompilación cero!**

Type Object: Desventajas

- Sólo sirve cuando las clases se **diferencian por los datos**
  - Tal como lo hemos definido, el **Type Object** es especialmente útil cuando hablamos de **diferenciar clases por sus datos**
  - Si quisiéramos, podríamos **implementar derivadas** del **Type Object** que también aportaran código diferenciado...
    - *... pero entonces ¿para qué hemos implementado el Type Object?*
  - **¡O utilizar Bytecode!** Pero ya hemos dicho que esto es **especialmente difícil de hacer**
- El caso más útil, cuando **cargamos datos desde el fichero XML**, es complejo de programar
  - Pero si encaja **vale realmente la pena**
- Se abren ciertas **dudas de diseño** importantes
  - ¿Puede un monstruo cambiar de tipo a mitad de partida? ¿Es eso viable/desable? ¿Cómo puedo evitar que ocurra?

Type Object: Notas Finales

- En lenguajes como **C o C++, Type Object** puede ser realmente útil
  - En **Commandos 2**, todos los objetos eran **Bicho** y cargaban sus datos desde ficheros de texto
- En **Unity**, y gracias a que **C# permite reflexión**, es mucho más fácil **extender tipos y no tener que recompilar** todo el proyecto
  - Utilizaremos para ello **Assemblies** y **Addressables**
- La manera más parecida en **Unity** de implementar el **Type Object** como lo hemos descrito aquí serían los **ScriptableObjects**