# PROGRAMACIÓN AVANZADA EN UNITY3D

# - Arquitectura, rendimiento y videojuegos -

Enrique Sainz para Level-Up

# Arquitectura del Software

- La Arquitectura del Software estudia la organización del código, no el propio código en sí

- Una buena arquitectura sería aquella en la que, a la hora de hacer un cambio en el código, bastase con tocar cuatro funciones sin interferir en el resto del proyecto
  - Esto también es aplicable a la arquitectura de **nuestras escenas** en **Unity3D**
  - ***Preguntadme por los "Scene Essentials" del GardenWorld***

# ¿Cómo hacemos un cambio en el Software?

- **Paso 1: Subir el código a nuestro cerebro de primate**
  - Para **entenderlo**, detectar el error o determinar las acciones necesarias para la mejora, y ponerlas en práctica

- ¿Hace falta subir todo el código?
  - **¡Todo no! Sólo lo relevante**

- Pero... ¿qué es lo relevante? ¿Y cómo sabremos lo que es relevante? ¿Y podemos **optimizar** este paso?

# A Desacoplar

- Podemos **desacoplar el código**, de modo que:
  - los cambios quedan **localizados en uno o dos módulos**
  - por tanto **sólo hay que subir esos módulos** a nuestro cerebro de primate
  - y **sólo habrá que cambiar esos módulos**, facilitando encontrar errores provocados por nuestros cambio

- En resumen, **desacoplar minimiza la cantidad de conocimiento que necesitamos tener en la cabeza** para progresar en nuestro trabajo

# Cómo Desacoplar

- Normalmente, desacoplaremos a través de interfaces comunes
  - En C++ utilizaremos herencia múltiple
  - En C# podemos utilizar una clase base común, o las interfaces específicas de C#
- En la mayoría de los casos esto será suficiente
- En otros casos, podemos utilizar otros sistemas, como eventos o componentes
  - **Ejemplo:** en Unity3D las físicas permiten colisionar contra cualquier objeto, independientemente cuál sea
  - **Ejemplo +:** si además queremos hacer una acción específica, podemos hacerlo a través de GetComponent, obteniendo un **componente concreto**...
  - **Ejemplo ++:** ...o incluso una interfaz...
  - **Ejemplo +++:** ... o incluso una clase base común a varios componentes

# El Coste de una Buena Arquitectura

- Si empezamos a desacoplar, debemos **desacoplar siempre**, o habremos desperdiciado nuestro esfuerzo
  - A la que dejas de mirarlo, el código se enmaraña solo muy rápidamente
- Si no vamos con ojo, empezaremos a desacoplar partes del código que después **no necesitan de tanta flexibilidad**
  - Crece la complejidad del código **a cambio de nada**...
  - ... y al final el cerebro de primate **se llena de una organización compleja**, en lugar de código útil
- **YAGNI:** *"You aren't gonna need it"*
  - Debemos valorar **si realmente lo vamos a necesitar**
  - De hecho, vale la pena **refactorizar -** añadir las interfaces **cuando realmente las vayamos a necesitar**
  - Eso sí: en caso de sospecha, **estructurar el código de tal modo que después sea fácil de refactorizar**

# Arquitectura: Exempli Gratia

- En el libro **Clean Code**, **Robert C. Martin** comparaba el código estructurado con un taller mecánico
  - ¿Como es más fácil **encontrar** las piezas que necesitamos?
    - Cuando están todas **esparcidas y mezcladas** por el suelo
    - Cuando están todas **organizadas y ordenadas** en sus cajones

- **Robert Nystrom** nos enseña que, si se nos va de las manos, podemos acabar teniendo las piezas organizadas en varios almacenes **IKEA**
  - Sin duda hay **un sitio para cada cosa**, pero también mucho espacio vacío
  - Y tener que caminar del pasillo 12 al pasillo 20 - o incluso ir de Hospitalet a Badalona a buscar lo que necesitamos acaba consumiendo más tiempo que buscarlo en medio del caos

- Lo ideal es que nuestro taller (**nuestra arquitectura**) tenga únicamente el **espacio necesario**
  - Ni más grande ni más pequeño de lo estrictamente necesario **para nuestras necesidades actuales**

# Rendimiento y Velocidad

- Una crítica importante a la Arquitectura es que daña el rendimiento de nuestro juego
  - Esto en parte es cierto: la flexibilidad se implementa con clases y métodos virtuales, que son en sí más caros de llamar que los no virtuales
    - Hablamos de a nivel de CPU - VTables, JMPs, LJMPs...

- Pero ¿de qué sirve el rendimiento si no podemos iterar lo suficientemente rápido?
  - La flexibilidad proporciona herramientas para que el diseñador **pruebe y ajuste** los resultados de poner en marcha sus ideas - *“ni siquiera **Will Wright** (SimCity, Spore, The Sims) puede crear un juego ya **equilibrado** sobre el papel”*

# Rendimiento y Velocidad... y Diversión

- ***“Mi experiencia, sin embargo, es que es más fácil hacer que un juego divertido sea rápido, que no hacer que un juego rápido sea divertido”***
  - **Robert Nystrom,**
    **destructor de *“Graphic Whores”*,**
    **y tío sensato, así en general**

(Por rápido entendemos en cuanto a rendimiento)

# Rendimiento y Velocidad... en la actualidad ( I )

¡ESTO NO ESTÁ EN EL LIBRO!

- Desde la perspectiva de los años 2000, cualquier parte de código que no tuviera un propósito funcional se veía como código superfluo

- Pero incluso en ese momento los **compiladores** permitían reducir el impacto de ese código superfluo con herramientas como el **inlining** o la **optimización de código máquina**

- Además, los ordenadores comenzaban a implementar técnicas como:
  - **Branching Prediction** (Procesadores Pentium II y Pentium III)
    - Permitían adelantarse al resultado de código condicional antes de que llegase el dato a ser evaluado
  - **Procesado matricial** (chips MMX)
    - Permitían paralelizar tratamiento gráfico (son los precursores de las tarjetas gráficas)
  - **Hyper Threading** (dos núcleos virtuales por cada núcleo real)
    - Aprovechando los tiempos muertos de un proceso, se daba paso al otro, manteniendo el núcleo un 30% más ocupado

# Rendimiento y Velocidad... en la actualidad ( II )

¡ESTO NO ESTÁ EN EL LIBRO!

- Posteriormente la popularización de las **tarjetas gráficas** traslada gran cantidad del trabajo de la CPU a la GPU
  - El pintado de pixels, básicamente, lo que podía suponer un **90% del fotograma**
  - ¿Por qué no utilizar parte de ese tiempo ganado en **código menos eficiente pero más barato de realizar** ?
    - ¡A la larga! ¡Nada de chapuzas!

- En la actualidad, los **motores gráficos** como Unity llevan al **extremo** la Arquitectura sin apenas compromiso con el rendimiento

# Rendimiento y Velocidad... y Diversión

¡ESTO NO ESTÁ EN EL LIBRO!

- *“Al final del cuento se descubrió que lo valioso no era tratar de optimizar el ordenador, sino el tiempo de los desarrolladores”*
  - **Enrique Sainz,**
    **aficionado a ver YouTubes de Ingeniería y Economía,**
    **y programas del tío escocés de la gorra que va**
    **con la camioneta comprando antigüedades**

# Bondades del Mal Código

- Hay un momento y un lugar para **diferentes estilos de código**
  - ¿También para las “chapus”? ¡También para las “chapus”! ¡Vivan los mancos!
- **Si tu código va a vivir más que tu en la empresa: Clean Code**
  - O, en palabras de **Robert C. Martin**, deja el código más limpio de lo que te lo encontraste
- Pero si estás probando **si una idea simplemente funciona**, Arquitecturar de más es tiempo que te arriesgas a perder
- **¡Cuidado con los prototipos!** Si funcionan, es difícil convencer al jefe de que hay que volver a escribirlos
  - **Truco:** prototipar en un lenguaje diferente al que se utilice en la empresa
  - Con **Unity** este truco no funciona... bueno siempre puedes hacer el código con cajitas...
    - ... salvo que te llames **Sergio Ossimo** y tu jefe tenga más cara que espalda

# Bondades del Mal Código... ¿proveedor o cliente?

¡ESTO NO ESTÁ EN EL LIBRO!

- Entendemos como código **puramente cliente** aquel código que no es utilizado por otros
  - Este es normalmente el **código específico de nuestro juego**, que únicamente existirá en este juego, y que ningún otro juego utilizará jamás
  - Siempre que nuestro desarrollo **no vaya a ser muy largo en el tiempo**, podemos permitirnos ser un poco menos cuidadosos
    - Cuanto **más largo el plazo, o mayor el proyecto**, más necesidad de que la arquitectura ayude a nuestro **cerebro de primates**
- En cambio, el **código proveedor** es aquel que se utiliza desde otros proyectos
  - **Librerías, utilidades, herramientas, y, especialmente, motores**, tienen que estar escritos con un acento mucho mayor en su Arquitectura
  - Sobretodo si pretenden ser usados **durante muchos años**
    - A mayor uso, normalmente mayor beneficio

# Alcanzando el Equilibrio

- Tenemos diferentes **fuerzas** en juego:
  - Queremos una **Arquitectura** que facilite entender el código a lo largo de todo su ciclo de desarrollo
  - Queremos **alto rendimiento** durante la partida
  - Queremos implementar las **características** de los juegos de hoy día, y queremos hacerlo **rápido**

- La respuesta **no es fácil**, ya que **estirar** una de esas fuerzas suele implicar **encoger** las otras dos

- **¡Bienvenidos a los malabares de la producción!**
  - Como en cualquier carrera con un poco de interés, la gracia está en **saber elegir adaptándose al contexto** del proyecto... a tiempo real

# Simplicidad

- Si hay una manera de equilibrar las fuerzas antes mencionadas esta es la simplicidad: **escribir la solución más simple y limpia a cada problema**

- Para ello es imprescindible **elegir las estructuras de datos más simples y funcionales**
  - Sobretodo no duplicar datos

- En cuanto al código, hay que elegir **generalizar** frente a tratar caso por caso
  - **Eg. Caso por caso:** detección de colisión entre pelota y jugador, entre pelota y portería, entre portería y jugador, entre todos los anteriores y el juez de línea...
  - **Eg. Generalización:** motor de físicas
  - La idea es sencilla de entender, pero la realidad es que se nos suele pedir que realicemos un módulo para **solucionar una serie de casos** - y es **tentador** solucionarlos cada uno por separado

# Al Ataque - Conclusiones

- La **abstracción** y el **desacoplamiento** hacen que programar sea más **rápido y fácil**, pero **no hay que perder el tiempo** donde la flexibilidad no sea necesaria
- **Preocúpate del rendimiento** del juego a lo largo de todo el desarrollo, pero **toma las decisiones de optimización más "calcificantes" lo más tarde posible**
- **Explora el diseño** de tu juego con celeridad, pero no tanta que dejes una **maraña** tras de tí: tendrás que vivir con ésta el resto del desarrollo
- Si vas a descartar código, **no te molestes en dejarlo bonito**
- Y, lo más importante: **si quieres hacer algo divertido, diviértete haciéndolo**