Programar con Estilo

En este documento vamos a establecer algunas normas que todos debemos cumplir a la hora de escribir código. Inicialmente, este documento también servirá para comprender algunos de los conceptos más avanzados del *framework* que vamos a utilizar para desarrollar el juego.

El patrón *singleton*

En ingeniería de software, *singleton* o instancia única es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. De este modo, nos evitaremos los infames punteros a la clase *Game* o podremos implementar con facilidad gestores de *assets* como el *TextureManager*.

Data-driven design

Por el tipo de juego que queremos hacer, sería recomendable que la creación de los estados y los objetos de juego viniese determinada por un archivo externo (XML, en este caso). De este modo, si por ejemplo queremos generar un nivel nuevo con sus respectivos enemigos, ítems, etc., no tendremos que crear un nuevo estado de juego, sino que nos valdrá con leer el correspondiente archivo de texto. Mantener las clases lo suficientemente genéricas como para que su estado lo determinen los datos que leemos de archivos externos es lo que se conoce como *data-driven design*.

Para explotar todo el potencial de este patrón de diseño, necesitaremos implementar una fábrica de *Game Objects*, una clase que tiene como tarea crear objetos de juego. Gracias al uso de fábricas distribuidas, podemos mantener dinámicamente los tipos de nuestros objetos; así, cada vez que registremos un nuevo tipo en nuestra fabrica, podremos empezar a crear instancias de ese objeto a través de la misma. La clase *LoaderParams* esta pensada para agrupar todos los atributos comunes de los objetos de juego, de modo que a la hora de cargar los datos del archivo de texto se simplifique la inicialización de los mismos.

Gestor de *assets* e *input*

Los métodos que hemos utilizado hasta ahora para gestionar imágenes, fuentes y audio son bastante pobres y, sobre todo, no escalan. Necesitamos gestionar nuestros *assests* de una manera rápida y sencilla, y por ello es necesario crear clases que se encarguen de esta tarea. Estos *asset managers* son clases diseñadas con el patrón *singleton* en mente que se caracterizan por contener un diccionario, además de algunas funcionalidades propias (como *render* en el caso del *TextureManager* o *play* en el caso del *AudioManager*). El diccionario de los gestores mapea los recursos a través de *strings*, de modo que si a lo largo del código queremos acceder a algún *asset* en concreto, sólo tenemos que recordar su ID. También disponemos de una clase *InputHandler* que controla todo el input de usuario, vía teclado, ratón y joystick (DualShock4); las funcionalidades específicas deben definirse en cada estado de juego.

La clase *Vector2D*

Esta clase está diseñada para facilitar todos los cálculos con estructuras de pares. Redefine varios operadores y es útil para agrupar valores: posiciones, velocidades, aceleraciones, direcciones, etc. Un ejemplo de como funcionaría el movimiento aplicando vectores sería muy similar a lo que vimos con Unity: *m_position += m_velocity.*

Variables miembro, locales y constantes

Los atributos de clase, o variables miembro, son características de cada una de las instancias de dicha clase. Por claridad a la hora de utilizarlas en los archivos de implementación (.cpp), se recomienda emplear una nomenclatura clara que indique su *status*:

Prefijo	**Significado**	**Ejemplo**
m	*member (v. miembro)*	*m_position*
p	*pointer (puntero)*	*m_pTexture*
b	*boolean (booleano)*	*m_bRunning*
s	*static (estática)*	*s_count*

Las variables locales, es decir, aquellas que definimos y utilizamos dentro del *scope* de una función o método, no se atienen a esta nomenclatura; así podemos diferenciarlas de las variables miembro. Las constantes se definen antes de la declaración de clase, y recomendamos escribirlas en mayúsculas: *const int WINDOW_WIDTH = 800.*

Directrices sobre *includes*

Las librerías básicas necesarias para el proyecto ya se encuentran añadidas a las rutas adecuadas. Cualquier adición o modificación sobre estas librerías o sus rutas se debe comunicar al resto del equipo para mantener la coherencia en el proyecto.

La utilización de la directriz *using namespace std* es una mala práctica en C++ que deberíamos tratar de evitar. Recordad que la librería estándar incluye algunas de las declaraciones mas habituales en nuestro código, por lo que deberemos acostumbrarnos a prefijarlas: *std::cout, std::endl, std::string, std::ofstream…*

Debemos mantener el código lo más libre de conflictos posible, por lo que si utilizamos algún *#include* debemos tener en cuenta lo siguiente:

1. El *#include* no ha sido realizado por alguna clase padre.
2. El *#include* no ha sido realizado por alguna de las cabeceras que ya hemos incluido.
3. Si el archivo sólo se va a utilizar en el archivo de implementación (.cpp), el *#include* sólo debe realizarse en dicho archivo, no así en la cabecera.
4. En caso de conflicto de inclusión circular se debe consultar con el resto del equipo la mejor manera de resolver el conflicto; tened en cuenta que una reestructuración de la jerarquía de clases nos afecta a todos.