MANUAL DE ESTANDARIZACIÓN DE PROYECTOS UNITY

ESTRUCTURA

➔ DE LA CARPETA DEL PROYECTO

- Cread esta estructura en la carpeta Assets:

3rdParty
_Nucleo
Editor
Gizmos
Plugins
Resources
Pruebas
Standard Assets

- "3rdParty" es donde se arrastran los assets que vienen del Asset Store. El 90% se podrán arrastrar, aunque no todos. Sabrás que un recurso no funcionará en una subcarpeta si recibe un error de "no se puede crear" desde el recurso (después de arrastrarlo a una subcarpeta), entonces el recurso debe permanecer en la raíz. Si importas un paquete que está almacenado disperso en varias subcarpetas de Assets, arrastra todo el material a la carpeta 3rdParty para limpiar la raíz.

- "_Nucleo" son todos los recursos que sabemos que definitivamente se están utilizando en el juego (lo distinguimos con un guión bajo). Dado que todos los recursos de Asset Store se instalarán en la raíz, los arrastraremos a "3rdParty" hasta que realmente queramos usarlos en nuestro juego. Luego los arrastraremos a la carpeta "_Nucleo" si realmente se están utilizando. La estructura interna de «_Nucleo» es la siguiente:

_Nucleo
- **_Animaciones**
- **_Arte**
  - **_Materiales**
  - **_Modelos**
    - **_Lugares**
    - **_Naturaleza**
    - **_Personajes**
    - **_Props**
  - **_Texturas**
- **_Audio**
  - **_Musica**
  - **_SFX**
  - **_Voces**
- **_Cielo**
- **_Código**
- **_Datos**
- **_Escenas**
- **_GUI**
- **_LineasTemporales**
- **_Terreno**

- "Editor" está en la carpeta principal porque no debería estar integrada en el juego. De forma predeterminada, Unity no crea nada de lo contenido en la carpeta "Editor", sino que lo reserva a recursos utilizados por el Editor.

- "Gizmos" permite agregar gráficos a la vista de escena para ayudar a visualizar detalles del diseño que de otro modo serían invisibles.

- "Plugins" → Unity compilará estos scripts en primer lugar, antes que el resto. Para que Unity no intente re-compilar todos los scripts del proyecto cada vez (incluso cuando solo hacemos un cambio), coloquemos los scripts a la carpeta "Plugins", pues así son ignorados en la recompilación automática. Si colocamos temporalmente la carpeta "_Core" y "3rdParty" a la carpeta "Plugins" ahorraremos mucho tiempo a lo largo del desarrollo del proyecto. Cuando hayamos terminado y esté listo para compilar, **GUARDEMOS UNA COPIA DE SEGURIDAD del proyecto a parte**, y luego arrastremos "_Core" nuevamente a la raíz, eliminando "3rdParty" porque lo que no esté contenido en _Core es que no se utiliza en el proyecto.

- "Resources" → Puede cargar recursos dinámicamente desde un script en lugar de crear instancias de recursos en una escena del juego. Para ello, colocamos los assets en una carpeta llamada Resources. Cargamos estos assets utilizando la función Resources.Load.

- "Pruebas" es una carpeta que nos servirá para testear cualquier cosa, como un banco de pruebas a parte del proyecto principal.

- "Standard Assets" → Cuando importamos un paquete de Standard Assets, Unity coloca los assets en esta carpeta. Además de contener los recursos, estas carpetas tienen un efecto en el orden de compilación de los scripts: se ejecutan primero, al igual que Plugins.

➔ <u>DE LA LISTA DE OBJETOS DE LA ESCENA</u>

- Cread esta estructura en la lista principal de objetos de cada escena:

  - **Gestores** //Controladores del flujo de todo el juego
    - GestorJuego
    - GestorNPCs
    - ...
  - ----------
  - **Configuración** //Todo lo que incluye la escena dinámica
    - CámaraPrincipal
    - Luces
      - LuzGlobal
      - Foco1
      - ...
    - Eventos
    - Samuel
    - NPCs
      - NPC1
      - NPC2
      - ...
    - ...
  - ----------
  - **Entorno** //Elementos de entorno estático

        - Terreno
        - Árboles
            - Árbol1
            - Árbol2
        - Casas
            - Casa1
            - Casa2
        - Calles
        - ...
    - ---
    - **Lienzos** //Elementos 2D sobre canvas
        - Principal
        - Secundario
        - Fondo
        - ...
    - ---
    - **Sistemas** // Objetos que usan DontDestroyOnLoad
        - SistemaAudio
            - Música
            - Sonidos
        - SistemaRecursos
        - ...

CONVENCIONES DE NOMENCLATURA

- En general, todo nombre escrito en castellano.

- Utilizaremos **PascalCase** para:
  - **Carpetas del proyecto.** Ejemplo: TexturasCasaSamuel
  - **Objetos del escenario**. Ejemplo: CamaraPrincipal
  - **Nombres de clases** del código. Ejemplo: MovimientosPersonaje
  - **Funciones** del código. Ejemplo: CalcularVelocidadAvance()

- Utilizaremos I+**PascalCase** para:
  - **Nombres de clases interface** del código. Ejemplo: IPersonajeNPC

- Utilizaremos **camelCase** para:
  - **Variables**. Ejemplo: numeroEntradasEnHabitacion

- Utilizaremos **_ + camelCase** para:
  - **Los campos privados** de un método o clase, para diferenciarlos de variables locales. Ejemplo: _estadoAnimacion

- Utilizaremos **MAYÚSCULAS** para:
  - Constantes. Ejemplo: MAXIMONUMEROVIDAS

- Utilizaremos **camelCase iniciado con verbo** para:
  - Booleans. Ejemplos: estaMuerto, tienePuntosDeHerida, esVisible...

- Escoger nombres claros, autoexplicativos y con toda la información. Ejemplo:
  En lugar de v, i, mPt, utilizar velocidad, contador, maximoPuntosSamuel

- En los nombres, no utilizar nunca tildes ni caracteres extraños. Ejemplos:
  AnadirLibro(), epocaDelAno...

- En las texturas, utilizaremos el **prefixo t_ + camelCase**. Ejemplos:
  t_troncoArbolCentral, t_carasDado...

- En los materiales, utilizaremos el **prefixo m_ + camelCase**. Ejemplos:
  m_troncoArbolCentral, m_carasDado...

- En los modelos, utilizaremos el **prefixo M_ + camelCase**. Ejemplos:
  M_troncoArbolCentral, M_carasDado...

- En los sonidos SFX, utilizaremos el **prefijo s_ + camelCase**. Ejemplos:
  s_lluviaSuave, s_pasosCesped…

- En las músicas, utilizaremos el **prefijo BSO_ + número de pista + PascalCase** para el nombre. Ejemplos:
  BSO_02_NombreDeLaPista2.ogg, BSO_04_NombreDeLaPista4.ogg…

- En las voces, utilizaremos el **prefijo v_ + la clave única interna del diálogo**. Ejemplos:
  v_SAMUEL_F5_AD_H0C1aF0.ogg, v_DAVID_F6_SA_H0C1F0.ogg...

CONVENCIONES DE ESTILO

- Los campos de una clase no deben ser públicos, sino **privados y serializables**:

  En lugar de utilizar

```
public float velocidad;
```

```
[SerializeField]
private float velocidad;
```

- Siempre que se pueda, utilizar **Patrones de diseño**:

  - **Singleton pattern** → garantiza que una clase tenga **solo una instancia** y proporciona un punto global de **acceso** a esa instancia.

```
public class Program
{
   public static void Main(string[] args)
   {
      Singleton singleton1 = Singleton.Instance;
      Singleton singleton2 = Singleton.Instance; //La misma clase devuelve la misma instancia

      singleton1.MetodoInstancia();
      singleton2.MetodoInstancia();

      // Ambos objetos referencian la misma instancia
      Console.WriteLine(singleton1 == singleton2); // Salida: True
   }
}
```

- **Observer pattern** → define una relación de dependencia uno a muchos, donde **varios observadores son notificados automáticamente** de cualquier cambio en el estado de un objeto.

```
public class Program
{
   public static void Main(string[] args)
   {
      Noticias servidorNoticias = new Noticias(); // Creamos el Subject (servidor de noticias)
      Cliente cliente1 = new Cliente("Cliente 1"); // Creamos Observer 1 (cliente 1)
      Cliente cliente2 = new Cliente("Cliente 2"); // Creamos Observer 2 (cliente 2)
      Cliente cliente3 = new Cliente("Cliente 3"); // Creamos Observer 3 (cliente 3)

      servidorNoticias.RegistrarObserver(cliente1); // Registramos Observer 1 en el Subject
      servidorNoticias.RegistrarObserver(cliente2); // Registramos Observer 2 en el Subject
      servidorNoticias.RegistrarObserver(cliente3); // Registramos Observer 3 en el Subject

     servidorNoticias.NuevaNoticia("Se ha anunciado un nuevo producto."); //Nueva noticia

      // Output esperado:
      // Cliente 1 recibió la noticia: Se ha anunciado un nuevo producto.
      // Cliente 2 recibió la noticia: Se ha anunciado un nuevo producto.
      // Cliente 3 recibió la noticia: Se ha anunciado un nuevo producto.
   }
}
```

- **Command pattern** → convierte una solicitud en un objeto, permitiendo parametrizar clientes con diferentes solicitudes, **encolar o registrar solicitudes** y soportar operaciones que **se pueden deshacer**.

```
public class Program
{
   public static void Main(string[] args)
   {
      Receiver receiver = new Receiver();        // Crear el receptor
      ICommand command = new ConcreteCommand(receiver); // Crear comando con receptor
      Invoker invoker = new Invoker();           // Crear el invocador y configura el comando
      invoker.SetCommand(command);
      invoker.EjecutarComando(); // Ejecuta acción en Receiver a través del invocador
   }
}
```

- **Component pattern** → permite construir estructuras jerárquicas de objetos utilizando una interfaz común, facilitando la **composición de objetos simples en estructuras más complejas**.

```
public class Program
{
   public static void Main(string[] args)
   {
      Compuesto compuesto = new Compuesto();        //Se crea la clase compuesto
      compuesto.Agregar(new Hoja("Hoja 1"));        //Se agrega hoja 1
      compuesto.Agregar(new Hoja("Hoja 2"));        //Se agrega hoja 2

      Compuesto compuesto2 = new Compuesto();       //Se crea otra clase compuesto2
      compuesto2.Agregar(new Hoja("Hoja 3"));       //Se agrega hoja 3
      compuesto2.Agregar(new Hoja("Hoja 4"));       //Se agrega hoja 4

      compuesto.Agregar(compuesto2);                //Los combina como más módulos

      compuesto.Mostrar();                          //Salida: combinación de las 4 hojas
   }
}
```

- **Flyweight pattern** → reduce el uso de memoria **compartiendo tanto como sea posible los datos que son comunes** a múltiples objetos.

```
public class Program
{
   public static void Main(string[] args)
   {
      FlyweightFactory factory = new FlyweightFactory();
      IFlyweight flyweight1 = factory.ObtenerFlyweight("tipo1");
      IFlyweight flyweight2 = factory.ObtenerFlyweight("tipo2");
      IFlyweight flyweight3 = factory.ObtenerFlyweight("tipo1"); //Se crea otro tipo1 distinto

      flyweight1.Mostrar(); // Se comparte el Flyweight "tipo1"
      flyweight2.Mostrar(); // Flyweight único "tipo2"
      flyweight3.Mostrar(); // Se comparte el Flyweight "tipo1"
   }
}
```

- **State pattern** → permite a un objeto **alterar su comportamiento cuando cambia su estado interno**, sin cambiar la declaración de sus métodos.

```
public class Program
{
   public static void Main(string[] args)
   {
      Contexto contexto = new Contexto();       // Se crea como clase común Contexto
      contexto.RealizarAccion();                // Salida: Ejecuta acción en Estado Inicial
      contexto.SetState(new EstadoIntermedio()); // Cambia a clase EstadoIntermedio
      contexto.RealizarAccion();                // Salida: Ejecuta acción en Estado Intermedio
      contexto.SetState(new EstadoFinal());     // Cambia a clase EstadoFinal
      contexto.RealizarAccion();                // Salida: Ejecuta acción en Estado Final
   }
}
```