Unity - Tema 8
Salvado y Carga de datos de juego

Scripting III

Javier Alegre Landáburu
javier.alegre@u-tad.com

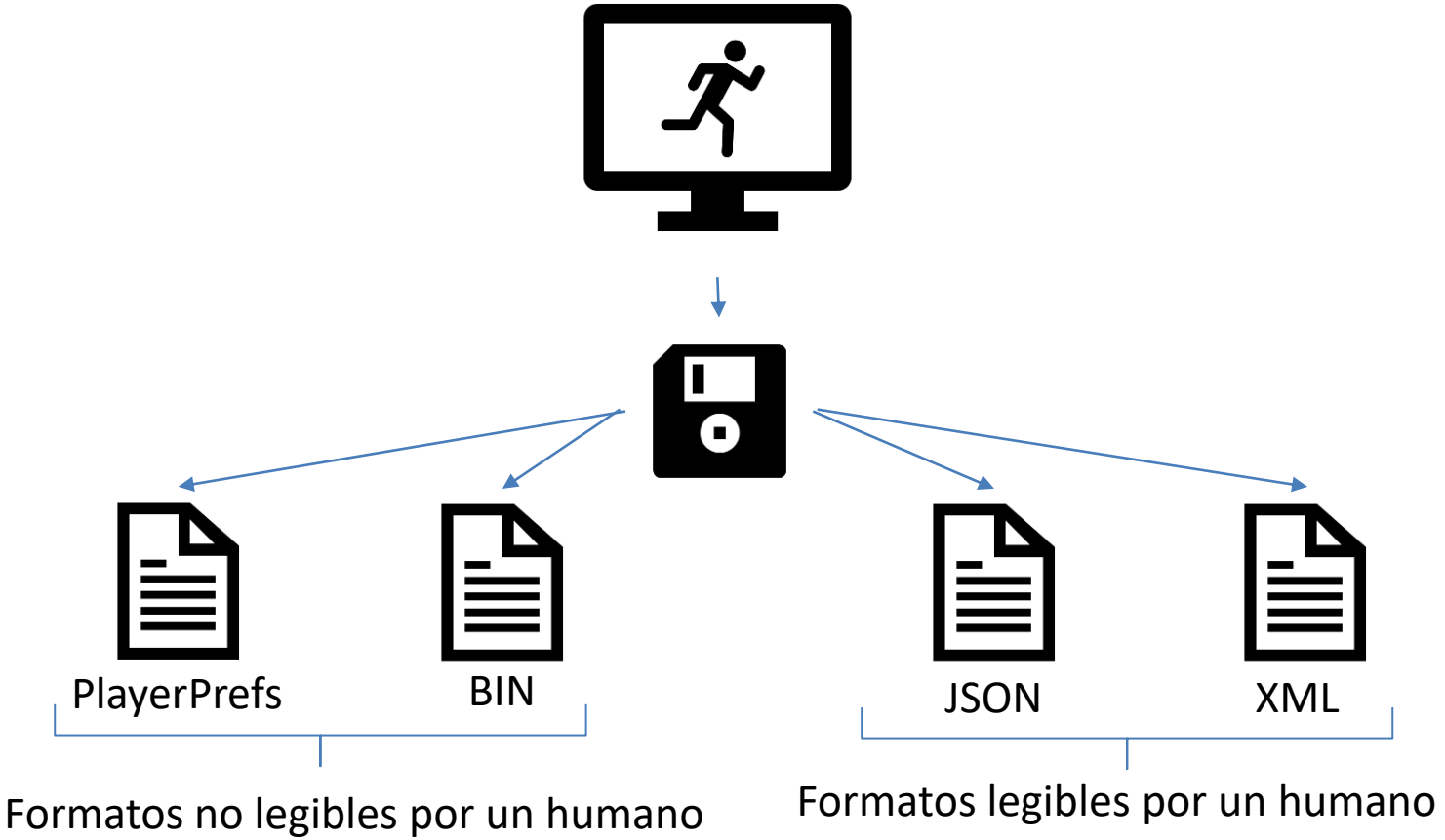
Resumen del tema

- ¿Cómo se guardan los datos de juego?
- ¿Qué tipos de ficheros podemos crear?
- ¿Se puede cargar/salvar propiedades de los objetos de la escena de manera sencilla?

Formas de almacenar datos

- Los datos de juego se guardan desde y se cargan en ficheros.

- Con Unity tenemos la opción de usar su propio sistema de guardado (*PlayerPrefs*) o de generar un fichero de datos y cargar/guardar en él (Binario, JSON, XML).



Player Prefs

- Sistema propio de Unity para cargar y guardar datos.
- Cada dato asociado a una clave.
- Sólo se pueden guardar variables de tipo int, float y string.

```
public class SaveableObject : MonoBehaviour
{
    public float  m_money;
    public int    m_points;
    public string m_playerName;

    void Save ()
    {
        PlayerPrefs.SetFloat ("Money",  m_money);
        PlayerPrefs.SetInt   ("Points", m_points);
        PlayerPrefs.SetString("Name",   m_playerName);
    }

    void Load ()
    {
        PlayerPrefs.GetFloat  ("Money",  0);
        PlayerPrefs.GetInt    ("Points", 0);
        PlayerPrefs.GetString ("Name",   "NoName");
    }
}
```

Datos

```
public class Player
{
    public string m_playerName;
    public float  m_gold;
    public float  m_wood;
    public float  m_iron;
    public float  m_stone;
    public int    m_victoryPoints;
    public int    m_battlePoints;

    public List<Card> m_cardList;
    public List<Item> m_itemList;
}
```

```
public class Card
{
    public string  m_cardName;
    public float   m_cost;
    public string  m_description;
    public int     m_type;
    public Sprite  m_sprite;
    public Color   m_color;
}
```

- ¡¡Pero si yo tengo un montón de datos!!

```
public class Item
{
    public string  m_itemName;
    public float   m_power;
    public string  m_description;
    public Sprite  m_sprite;
}
```

Ficheros

- Con tanta información, sería bastante complejo hacer el cargado/guardado de datos en *PlayerPrefs*.

- Vamos a crear un fichero para está tarea.

- El fichero podrá ser información en bruto (binario) o información legible por humanos y otros programas (XML y JSON).

- Sea del tipo que sea lo primero es añadir *[System.Serializable]* en cada clase que queremos serializar.

- Nosotros queremos serializar la clase *Player*, que contiene una lista de Cartas (Clase *Card*) e Items (Clase *Item*).

- Necesitamos añadir *[System.Serializable]* a todas para que la clase Player sea serializable. No se puede serializar una clase que contiene otras no serializables.

Ficheros

```
[System.Serializable]
public class Player
{
    public string m_playerName;
    public float  m_gold;
    public float  m_wood;
    public float  m_iron;
    public float  m_stone;
    public int    m_victoryPoints;
    public int    m_battlePoints;

    public List<Card> m_cardList;
    public List<Item> m_itemList;
}
```

```
[System.Serializable]
public class Card
{
    public string  m_cardName;
    public float   m_cost;
    public string  m_description;
    public int     m_type;
    public Sprite  m_sprite;
    public Color   m_color;
}
```

```
[System.Serializable]
public class Item
{
    public string  m_itemName;
    public float   m_power;
    public string  m_description;
    public Sprite  m_sprite;
}
```

- Creamos un pequeño script que en su Start crea un jugador y lo guarda para poder probar nuestra estructura de datos serializable.

Ficheros

```
[System.Serializable]
public class Card
{
    public string  m_cardName;
    public float   m_cost;
    public string  m_description;
    public int     m_type;
    public Sprite  m_sprite;
    public Color   m_color;
}
```

```
[System.Serializable]
public class Item
{
    public string  m_itemName;
    public float   m_power;
    public string  m_description;
    public Sprite  m_sprite;
}
```

- Si intentamos serializar estas clases tenemos un problema y es que tanto *Card* como *Item* contienen objetos de la clases Sprite y Color de Unity.

- Estas clases no son serializables y no puedo meterme en el código de unity para cambiarlo.

- En el caso del Sprite puedo guardar un *string* con su nombre y usarlo para cargar el Sprite en la carga de datos.

- En el caso del color puedo guardarlo en un Vector3 o con 3 variables numéricas.

Ficheros

Creamos un jugador para poder tener datos que guardar

```
public class SaveableObject : MonoBehaviour
{
    private Player m_player;

    void Start()
    {
        GenerateNewPlayer ();
        SaveBin           ();
    }

    void SaveBin           ()...

    void GenerateNewPlayer ()...
}
```

```
void GenerateNewPlayer ()
{
    // Creamos datos del jugador
    m_player                 = new Player();
    m_player.m_playerName    = "NoName";
    m_player.m_gold          = 0;
    m_player.m_wood          = 0;
    m_player.m_iron          = 0;
    m_player.m_stone         = 0;
    m_player.m_victoryPoints = 0;
    m_player.m_battlePoints  = 0;
    m_player.m_cardList      = new List<Card>();
    m_player.m_itemList      = new List<Item>();

    // Creamos datos de las cartas
    Card auxCard = new Card();
    auxCard.m_cardName = "Poison Arrows";
    auxCard.m_cost     = 0;
    //... el resto de características.
    m_player.m_cardList.Add(auxCard);

    // Creamos datos de los items
    Item auxItem = new Item();
    auxItem.m_itemName = "Increase Power";
    auxItem.m_power    = 50;
    //... el resto de características.
    m_player.m_itemList.Add(auxItem);
}
```

Guardado Binario

```
public class SaveableObject : MonoBehaviour
{
    private Player m_player;

    void Start()
    {
        GenerateNewPlayer ();
        SaveBin           ();
    }

    void SaveBin          ()...

    void GenerateNewPlayer ()...
}
```

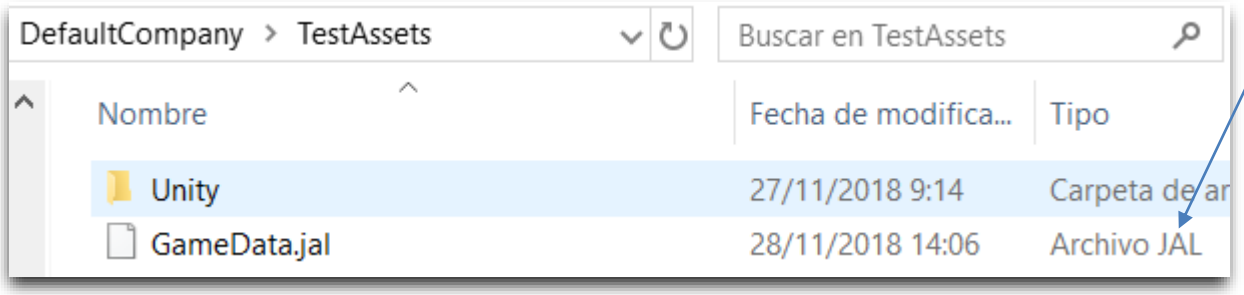
```
void SaveBin           ()
{
    BinaryFormatter bf    = new BinaryFormatter();
    FileStream      file = File.Create(Application.persistentDataPath + "/GameData.jal");
    bf.Serialize(file, m_player);
    file.Close();
}
```

El método de guardado es sencillo.

Usamos la clase *BinaryFormatter* para serializar la clase Player.

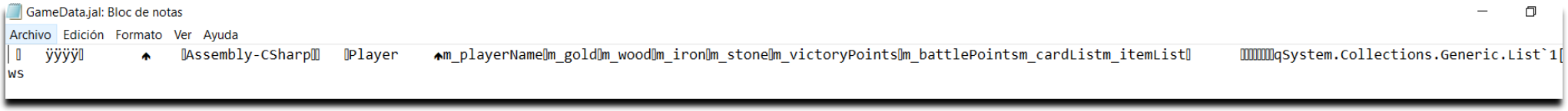
Usamos la clase *FileStream* y *File* para crear el fichero.

El fichero puede tener la extensión que yo quiera. En mi caso he puesto .jal que son mis iniciales.



Guardado Binario

Se crea un fichero que no es legible/modificable por el usuario.



Carga Binaria

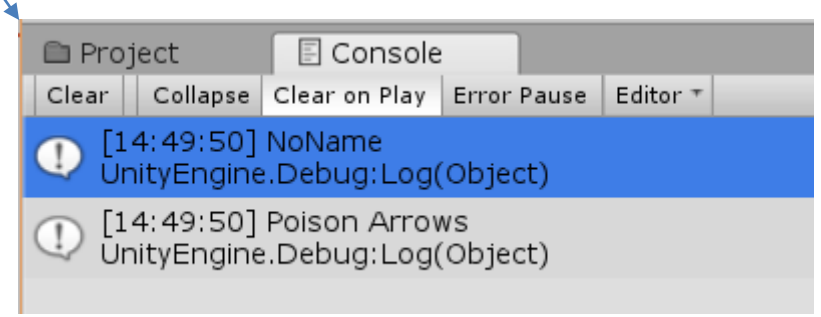
Para probar que se puede cargar el fichero, vamos a modificar nuestro script para que lo cargue al inicio.

Comentamos los métodos de crear nuevo jugador y guardar en el *Start* y añadimos el método LoadBin.

```
public class SaveableObject : MonoBehaviour
{
    private Player m_player;

    void Start()
    {
        //GenerateNewPlayer ();
        //SaveBin           ();
        LoadBin             ();
    }

    void SaveBin          ()...

    void LoadBin          ()...

    void GenerateNewPlayer ()...
}
```

Carga Binaria

```
void LoadBin           ()
{
    BinaryFormatter bf   = new BinaryFormatter();
    FileStream      file = File.Open(Application.persistentDataPath + "/GameData.jal", FileMode.Open);
    m_player             = (Player)bf.Deserialize(file);

    Debug.Log(m_player.m_playerName);
    Debug.Log(m_player.m_cardList[0].m_cardName);
}
```



XML

- XML es un estándar que estructura el intercambio de información entre las diferentes plataformas (una serie de reglas de cómo va a escribir la información en un fichero).

- Los datos guardados son legibles por un ser humano.

- Como esta muy extendido tenemos librerías que nos permiten cargar y guardar ficheros en este formato para casi todos los lenguajes y plataformas.

GameData.xml: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
<?xml version="1.0" encoding="Windows-1252"?>
<Player xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <m_playerName>NoName</m_playerName>
  <m_gold>0</m_gold>
  <m_wood>0</m_wood>
  <m_iron>0</m_iron>
  <m_stone>0</m_stone>
  <m_victoryPoints>0</m_victoryPoints>
  <m_battlePoints>0</m_battlePoints>
  <m_cardList>
    <Card>
      <m_cardName>Poison Arrows</m_cardName>
      <m_cost>0</m_cost>
      <m_type>0</m_type>
    </Card>
  </m_cardList>
  <m_itemList>
    <Item>
      <m_itemName>Increase Power</m_itemName>
      <m_power>50</m_power>
    </Item>
  </m_itemList>
</Player>
```

XML

- El guardado es muy sencillo, sólo tenemos que utilizar el *XMLSerializer*.

```
void SaveXML           ()
{
    XmlSerializer serializer = new XmlSerializer(typeof(Player));
    FileStream    file = File.Create(Application.persistentDataPath + "/GameData.xml");
    serializer.Serialize(file, m_player);
    file.Close();
}
```

- La carga de datos:

```
void LoadXml           ()
{
    XmlSerializer serializer = new XmlSerializer(typeof(Player));
    FileStream    file = File.Open(Application.persistentDataPath + "/GameData.xml", FileMode.Open);
    m_player = (Player)serializer.Deserialize(file);


    Debug.Log(m_player.m_playerName);
    Debug.Log(m_player.m_cardList[0].m_cardName);
}
```

JSON

- *JSON* es otro estándar que estructura el intercambio de información entre las diferentes.

- Los datos guardados al igual que en el caso del *XML* son legibles por un ser humano.

- También está muy extendido.

JSON

- Para el guardado sólo tenemos que utilizar la clase *JsonUtility*, que crea un *string* con estructura JSON y los datos del *Player*.

- Después escribimos el *string* en disco usando File.

```
void SaveJSON           ()
{
    string      json = JsonUtility.ToJson(m_player);
    File.WriteAllText(Application.persistentDataPath + "/GameData.json", json);
}
```

- La carga de datos:

```
void LoadJSON           ()
{
    string json = File.ReadAllText(Application.persistentDataPath + "/GameData.json");
    m_player = JsonUtility.FromJson<Player>(json);

    Debug.Log(m_player.m_playerName);
    Debug.Log(m_player.m_cardList[0].m_cardName);
}
```

DataManager

Hemos visto como guardar/cargar las estructuras de datos que usamos para el juego.

Es normal tener un DataManager de tipo Singleton que tenga esas estructuras y tenga sus métodos de guardado y carga de datos.

En la pantalla de Loading puedo mandar al DataManager que cargue los datos y luego haga la carga asíncrona de la siguiente escena.

El GameManager se encargará al inicio de cargar la escena de juego de manera que empiece en el punto en que quedo el juego en el momento de salvar.

GameObjects

¿Y en mi juego el usuario crea/modifica/destruye el escenario y por tanto modifica los *GameObjects* de la escena?

Tendremos que crear estructuras de datos para ello y comunicar de alguna manera esos *GameObjects* con el DataManager para que al hacer *Save* se guarden sus propiedades.

Vamos a ver una manera rápida y cómoda de hacerlo usando lo aprendido en la asignatura.

Lo primero es crearnos una estructura de datos para guardar los datos que queremos de un *GameObject*. En este caso vamos a guardar los valores de posición, rotación y escala.

```
[System.Serializable]
public class TransformData
{
    public string     m_name;
    public Vector3    m_position;
    public Quaternion m_rotation;
    public Vector3    m_scale;
}
```

GameObjects

Lo siguiente es crear un componente que añadiremos a todos los GameObjects que quiero guardar.

Este componente tendrá dos métodos, uno para convertir los valores del transform en un TransformData que podamos guardar y otro para leer el *TransformData* y setear los valores al GameObject.

```
public class SaveableMonoBehaviour : MonoBehaviour
{
    public virtual void SetData (TransformData data)
    {
        transform.SetPositionAndRotation(data.m_position, data.m_rotation);
        transform.localScale = data.m_scale;
    }

    public virtual TransformData GetData ()
    {
        TransformData data = new TransformData(transform, gameObject.name);
        return data;
    }
}
```

GameObjects

He creado un constructor para la clase *TransformData* al que se le pasan los datos que queremos guardar.

```
public class SaveableMonoBehaviour : MonoBehaviour
{
    public virtual void SetData (TransformData data)
    {
        transform.SetPositionAndRotation(data.m_position, data.m_rotation);
        transform.localScale = data.m_scale;
    }

    public virtual TransformData GetData ()
    {
        TransformData data = new TransformData(transform, gameObject.name);
        return data;
    }
}
```

```
[System.Serializable]
public class TransformData
{
    public string     m_name;
    public Vector3    m_position;
    public Quaternion m_rotation;
    public Vector3    m_scale;

    public TransformData () {}

    public TransformData (Transform data, string name)
    {
        m_name     = name;
        m_position = data.position;
        m_rotation = data.rotation;
        m_scale    = data.localScale;
    }
}
```

GameObjects

En el DataManager creamos un método que busque todos los *SaveableMonoBehaviour* de la escena y llame a sus *GetData* para que le devuelvan el *TransformData* con sus datos ya rellenos.

```
public void SaveGameScene ()
{
    SaveableMonoBehaviour[] saveGOVector = FindObjectsOfType<SaveableMonoBehaviour>();
    List<TransformData>     tranformList = new List<TransformData>();

    for (int i = 0; i < saveGOVector.Length; i++)
    {
        tranformList.Add(saveGOVector[i].GetData());
    }

    XmlSerializer serializer = new XmlSerializer(typeof(List<TransformData>));
    FileStream    file = File.Create(Application.persistentDataPath + "/GameSceneData.xml");
    serializer.Serialize(file, tranformList);
    file.Close();
}
```

GameObjects

Al llamar a esté método conseguimos un fichero que contiene una lista con todos los transformdata

```
<?xml version="1.0" encoding="Windows-1252"?>
<ArrayOfTransformData xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <TransformData>
    <m_name>SM_Tile_Hex_Column_01</m_name>
    <m_position>
      <x>-16.4226246</x>
      <y>3.52</y>
      <z>21.5950546</z>
    </m_position>
    <m_rotation>
      <x>0</x>
      <y>0.258820444</y>
      <z>0</z>
      <w>0.965925455</w>
      <eulerAngles>
        <x>0</x>
        <y>30.0001659</y>
        <z>0</z>
      </eulerAngles>
    </m_rotation>
    <m_scale>
      <x>1</x>
      <y>1</y>
      <z>1</z>
    </m_scale>
  </TransformData>
  <TransformData>
```

GameObjects

Para cargar los datos hacemos lo mismo.

1 – Leemos el fichero.

2 - Buscamos todos los gameobject con un componente SaveableMonobehaviour .

3 - Recorremos esa lista de SaveableMonobehaviour mirando si en nuestra lista cargada de TransformData se han guardado los datos de ese SaveableMonobehaviour (buscamos por el nombre del GameObject que hemos guardado)

4 - Llamamos al método SetData del SaveableMonobehaviour con los datos del TransformData encontrado.

GameObjects

Para cargar los datos hacemos lo mismo.