La igualdad de género está aún por conseguir a distintos niveles de la sociedad. Uno de ellos, es la paridad entre la población másculina y femenina en carreras de ingeniería. Este hecho está cambiando lentamente gracias a asociaciones como FemDevs, una asociación que recoge a las desarrolladoras de videojuegos para darles visibilidad.

Dentro de FemDevs se recogen las distintas ramas que existen en el desarrollo de videojuegos (diseño de juegos, arte y programación). Cada una de ellas tiene sus propias dinámicas para dar a conocer a sus integrantes. En el caso de las programadoras tenemos un reto para resolver, en este caso un sistema de diálogos.

Objetivo

Un sistema de diálogos permite que el jugador interactúe con los personajes existentes en el juego. El objetivo fundamental de este reto es crear un sistema de diálogos desde 0.

Sobre este objetivo primario me he propuesto que mi sistema de diálogos sea lo más integrado en el motor de videojuegos que uso habitualmente: Unity3D. Utilizando los conocimientos adquiridos estos dos años que llevo utilizándolo.

Así he dividido el desarrollo del reto en tres partes: backend, frontend y controller.

Backend

El backend en el desarrollo habitual de software es toda aquella parte que está conectada a la base de datos. Si lo extrapolamos al reto el backend es la parte que interactúa con el diálogo que hay que mostrar en pantalla.

Para hacer este backend lo más “Unity” posible se desarrolla dos tecnologías que he visto poco nombrar y menos utilizar en entornos indie: ScriptableObjects y Unity Editor.

Scriptable Object

Un Scriptable Object es un tipo de datos derivado de Object de unity que permite la creación de assets y almacenar datos en memoria, en mi gusto más eficientemente que los prefab y además es git-friendly.

Para almacenar un objeto como scriptable object primero tenemos que crear un tipo de datos serializable y que no herede de MonoBehaviour. En el caso de nuestro sistema diálogos este tipo de datos es un Diálogo:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[Serializable]
public class Dialog  {


    public string key;
    public string dialog;
    public string character;
    public string details_nextKey;
    public string details_prevKey;
    public string notes;


}
```

Ilustración 1: Clase serializable.

Nuestro diálogo contendrá:

- Una clave identificativa del diálogo.
- El diálogo en sí
- El nombre del personaje que habla
- La clave del diálogo que precede a este diálogo
- La clave del diálogo que sigue a este diálogo
- Notas que el escritor considere oportunas tener a la vista.

El Scriptable Object contendrá una lista de este tipo de datos y una función para realizar la búsqueda por clave de los diálogos.

```
public class DialogData : ScriptableObject {


    public List<Dialog> dialogSystem;


    public Dialog GetDialogByKey(string key) {
        return this.dialogSystem.Where<Dialog>(d => d.key.ToLower().Equals(key.ToLower())).FirstOrDefault();
    }
}
```

Ilustración 2: Contenido del scriptable object.

Para realizar la búsqueda por clave, se utiliza la librería Linq para encontrar el primer diálogo cuya clave en minúscula coincida con el nombre de la clave en minúscula. En caso de no encontrar la clave devolverá null.

La creación user-friendly de este scriptable object pasa por generar una ventana personalizada dentro de Unity.

Unity Editor

Crear una extensión de Unity es bastante sencillo solo hay que seguir los siguientes pasos:

- Heredar del tipo componente de Unity que queremos crear.
- Desarrollar el método estático Init
- Desarrollar el método OnGUI

En este caso el componente del que se hereda es *EditorWindow*.

El método Init es bastante común solo hay que obtener una ventana del componente que estamos creando tal que así:

```
[MenuItem("Window/Dialog Editor %#e")]
static void Init()
{
    EditorWindow.GetWindow(typeof(DialogGUI));
}
```

Ilustración 3: Método Init.

Lo que ya no es tan trivial es el desarrollo del método OnGUI porque hay que definir layouts, los componentes dentro de los mismos el aspecto… De todas formas si tienen un día aburrido pueden echarle un vistazo al método OnGUI en el enlace al repositorio que dejaré al final.

Al final la interfaz queda así:

Ilustración 4: Interfaz de Usuario para editar los diálogos

Como se puede ver, el Dialog Editor permite editar todos los campos que definimos antes en el tipo de datos Dialog.

Si hacemos clic en el botón Show Dialog List podemos ver en el Inspector una lista con todos los diálogos escritos hasta el momento con todas sus propiedades.

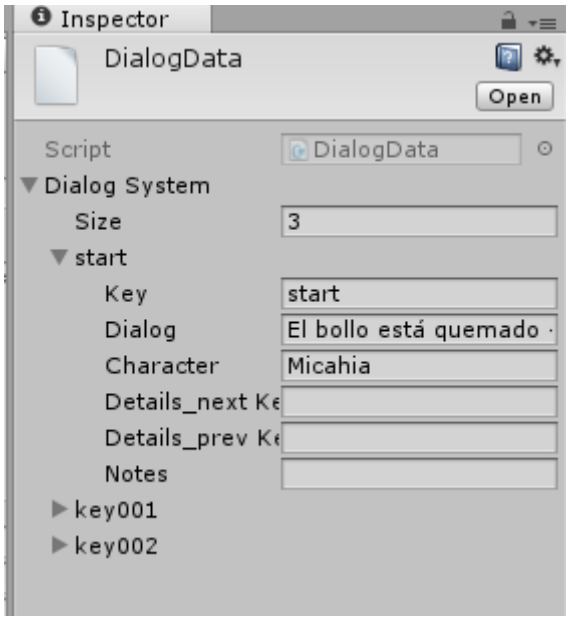


Ilustración 5: Vista en el inspector de las claves del diálogo

Ahora que tenemos desarrollada la base de datos de los diálogos y su interacción tanto con el usuario como con el código podemos pasar a hacer el frontend.

Frontend

El frontend es la parte del desarrollo que se encarga de mostrar el diálogo dentro del juego. En esta parte del desarrollo siempre intento aplicar los conocimientos que tengo en otras tecnologías para hacer un desarrollo lo más similar. En este caso intentaré que el frontend se este sistema de diálogos sea lo más parecido a los *usercontrols* de WinForms o Windows Presentation Fundation(WPF).

Tanto en WinForms como en WPF se puede crear la interfaz de usuario usando únicamente componentes gráficos así que empezaremos por esto.

Lo primero que hacemos es crear el Canvas donde se mostrarán todos los componentes de la interfaz gráfica y creamos un nuevo objeto vacío que llamaremos Dialog Control

Ilustración 6: Jerarquía dentro de la interfaz.

Dentro de DialogControl vamos a tener los botones tres botones de acciones, el texto donde se mostrará el diálogo y la imagen del bocadillo que estará debajo del diálogo. Añadiendo todo esto la jerarquía se nos queda de esta forma.

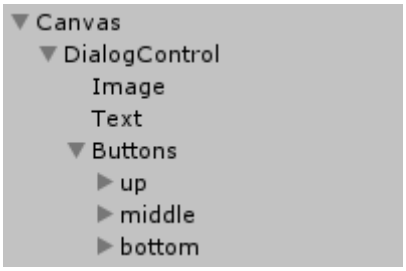
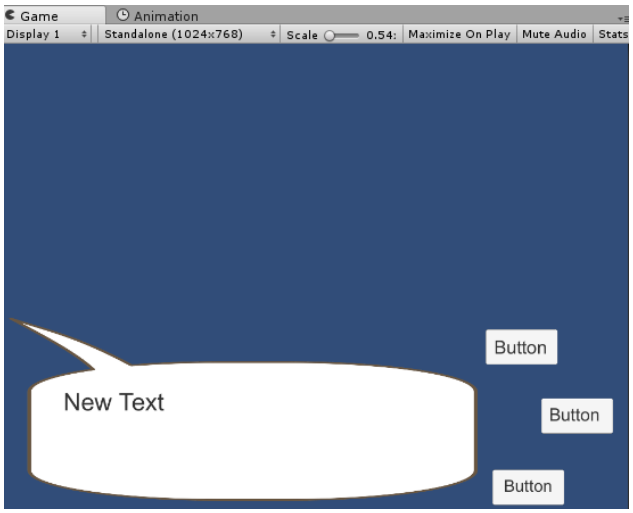


Ilustración 7: Jerarquía del DialogControl

Que nos devuelve una imagen como esta:



Una aclaración que tengo que hacer. Los botones están preparados para que cualquier texto se ajuste al espacio que tienen usando la propiedad “Best Fit”.

En WPF y WinForms todas las funciones para controlar a los descendientes suelen estar en el control principal por lo que se desarrolla un script Monobehaviour que nos permita realizar todas las funciones con solo acceder al componente *DialogControl*.

Las funciones que queremos realizar son las siguientes:

- Establecer el número de botones que aparecen en pantalla
- Establecer el texto de cada uno de los botones de forma individual
- Establecer la key a la que nos lleva cada uno de los botones.
- Establecer el texto a mostrar
- Modificar la posición del control en sí.
- Flip la imagen del control a derecha e izquierda.
- Recibir eventos al terminar el diálogo.

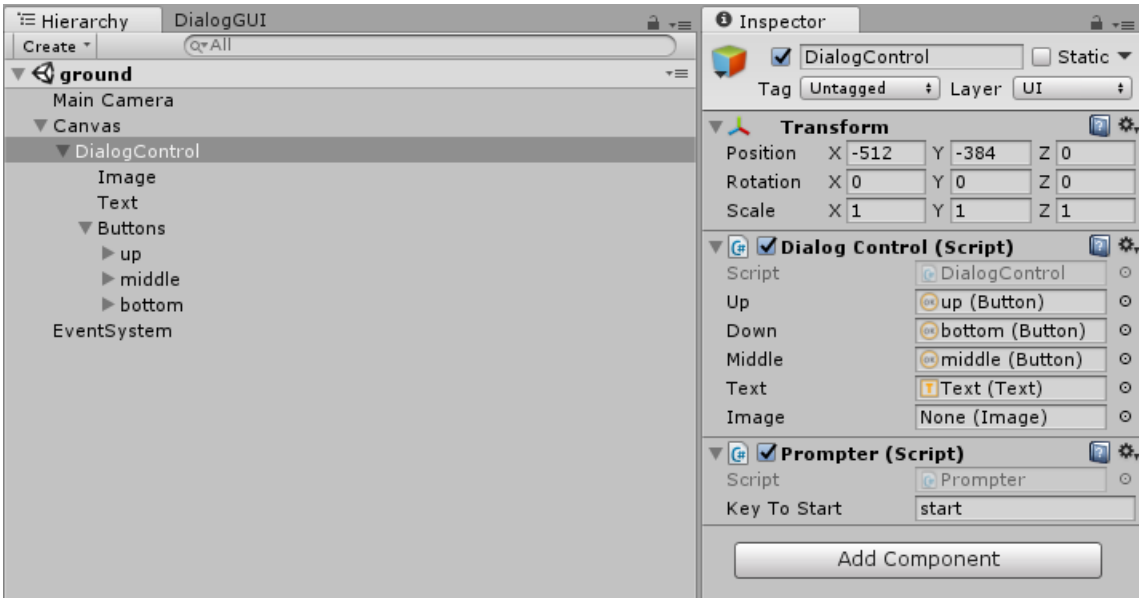
Además de estas funciones tenemos una función auxiliar que oculta los botones.

Con la elaboración de este control ya podemos mostrar toda la información de manera sencilla. Lo único que nos queda es conectar el frontend con el backend mediante el controller.

Controller

El controller es un término que viene del patrón de diseño Model-View-Controller también conocido como MVC. La finalidad del controller es realizar una acción, casi siempre iniciada por el usuario en la vista, lo que he denominado anteriormente como frontend.

La acción que queremos realizar aquí es leer del Scriptable object la clave/key que nos interesa y actualizar el *DialogControl* de acuerdo a la información de la clave. Para ello se elabora un script denominado Prompter que se encarga de desarrollar esta función, la única información que hay que darle al Prompter es la clave que tiene que buscar al inicio y ya se encargará de todo el solo.



Toda la funcionalidad del prompter la podemos ver resumida en la función prompt:

```
public void prompt() {

    ParserInfo parsed = Parser.parse(current.dialog);

    updateControl(parsed);

}
```

Esta función toma el dialogo actual, extrae la información de los botones, que viene codificada por el escritor, y actualiza el control de acuerdo a ella.

La información de los botones viene codificada de la siguiente manera:

dialogo<button 1 text key>[<button 2 text key>[<button 3 text key>]]

De forma que todo el diálogo va antes de definir el valor de los botones. Cabe la posibilidad de que no hayan botones.

También existe un segundo tipo de control en este desarrollo: el *ButtonBehaviour* este control se encarga de recibir la clave a la que va a actualizar todo el sistema de diálogos si se pulsa el botón y de actualizar el Prompter para que actualice el *DialogControl*.

El *buttonBehaviour* es un único script añadido como componente a cada uno de los botones, de esta forma nos ahorramos tener que escribir un control para cada opción dentro del diálogo.

Conclusiones

El desarrollo del sistema de diálogos tal y como se ha planteado me ha parecido una forma bastante directa de solucionar el problema sin dejar de lado al escritor.

Es cierto que el escritor preferiría otras herramientas como yarn que le permite poner condiciones al flujo del diálogo pero soy de esas personas que prefiere que todo esté centralizado, en este caso en Unity.

También estoy muy contenta de poder poner en contexto todo lo aprendido en estos dos últimos años de unity y sobre todo lo aprendido en este mes y medio que llevo trabajando en Rising Pixel.

¿Qué se queda en el tintero?

Si alguien ve en el código puede ver que también hay una corrutina preparada para motrar el texto letra a letra. No tuve tiempo de probarlo por lo que no lo he añadido a este texto.

También me he quedado con las ganas de probar la API de Test-To-Speech para reproducir el diálogo, pero implicaría utilizar .Net 4.X en la versión de Unity 2017 que está en versión experimental y cuando lo probé en su momento para el proyecto M.A.N.D.A.L.A no paraba de dar errores con la librería mscorlib. Así que tendré que probarlo con la versión 2017.2 en un futuro que espero que no esté tan lejano.