SISTEMAS DE DIÁLOGOS EN UNITY

REALIDAD Y ACCESIBILIDAD AUMENTADAS – CURSO 2020-2021

MARÍA FLÓREZ MIRANDA
– UO264446

INTRODUCCIÓN

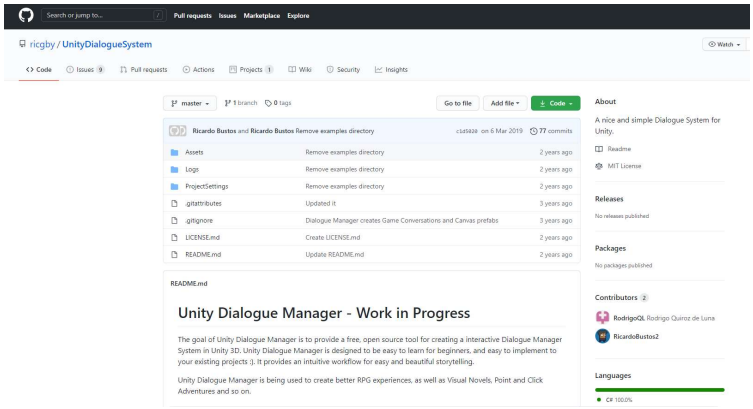
El objetivo de este proyecto ha sido la incorporación de un sistema de diálogos a una aplicación de juguete en Unity, de tal manera que pueda evaluarse su funcionamiento e incorporarse a proyectos de mayor envergadura. Se ha optado por probar dos tipos de sistemas de diálogos muy diferentes entre sí, uno partiendo de un proyecto creado y otro partiendo desde cero.

PRIMERA PARTE: UNITY DIALOGUE SYSTEM

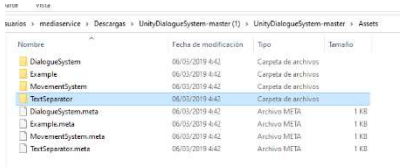
En esta primera parte, vamos a incluir en nuestro proyecto un sistema de diálogos ya existente, UnityDialogueSystem, creado por el estudio ScrollSomething como proyecto open source y disponible en el GitHub de su fundador (https://github.com/ricgby/UnityDialogueSystem). Cabe destacar que, a pesar de que supuestamente este sistema sigue en desarrollo, el repositorio parece abandonado, y la documentación es escasa. Conviene tomarse la molestia de investigar por nuestra cuenta los scripts y cómo interactúan las clases entre sí, ya que afortunadamente el código es bastante legible.

DESCARGA DEL SISTEMA DE DIÁLOGOS E INCORPORACIÓN DE ASSETS A NUESTRO PROYECTO

Lo primero que vamos a hacer es descargarnos el proyecto de GitHub, bien clonándolo en un repositorio local o bien descargándonos el ZIP con el código; por comodidad, es más recomendable la segunda opción, puesto que no vamos a necesitar todos los assets que vienen incluidos.

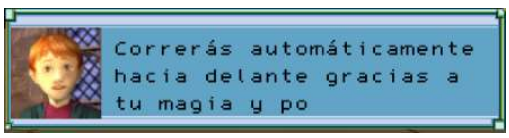


Una vez descargado el ZIP con el proyecto, tan solo necesitamos, de la carpeta “Assets”, el contenido de “DialogueSystem” y “TextSeparator”, que importamos en nuestro proyecto de la manera habitual.



CREACIÓN DE PERSONAJES

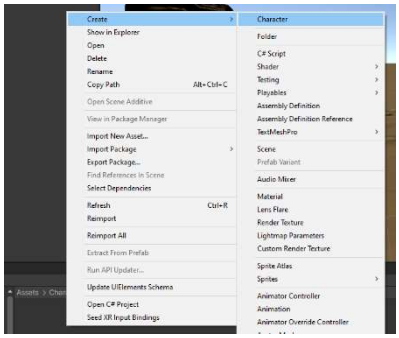
Adelantándonos un poco, una vez tengamos el sistema de diálogos implementado, tendremos algo similar a esto:

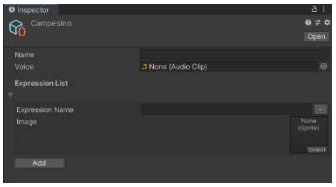


Como se puede ver, la caja de diálogo contiene la foto del personaje al que están asociadas las líneas de conversación que se muestran. Estos personajes hay que crearlos previo uso, lo cual es muy sencillo.

Lo primero de todo, es muy recomendable crearnos en nuestra carpeta de Assets una nueva carpeta para guardar todo lo relativo a estos personajes. En nuestro caso la hemos llamado DialogueSystemCharacters, pero realmente podemos darle cualquier otro nombre.

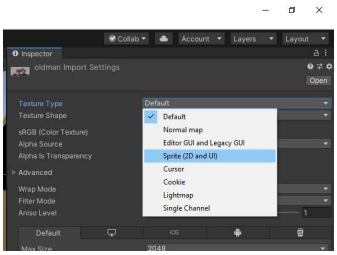
A continuación, como el DialogueSystem incorpora un script para crear assets de tipo "Character", simplemente en la carpeta recién creada nos vamos a Create>Character.



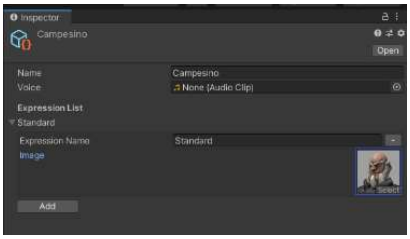


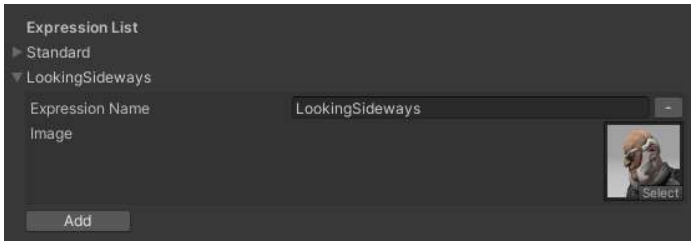
Si nos vamos al inspector veremos que cada Personaje tiene un nombre, una voz (un clip de audio, no es obligatorio) y una lista de expresiones. Cada expresión, a su vez, tiene un nombre que la identifica y una foto que represente esa determinada expresión, y que es la que aparece en cada momento en la caja de diálogos. Un personaje puede tener tantas expresiones como queramos.

Para crear una expresión, primero hemos de tener entre nuestros assets la imagen que deseemos asociarle. Vale cualquier formato de imagen, pero es recomendable que tenga una forma lo más cuadrada posible. A continuación, le cambiamos a esa imagen el Texture Type de Default a Sprite.

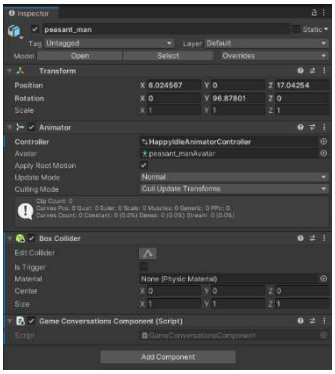


Ahora solo tenemos que añadir expresiones a la lista, darles un nombre característico y arrastrar la imagen que queramos para cada una al campo correspondiente:

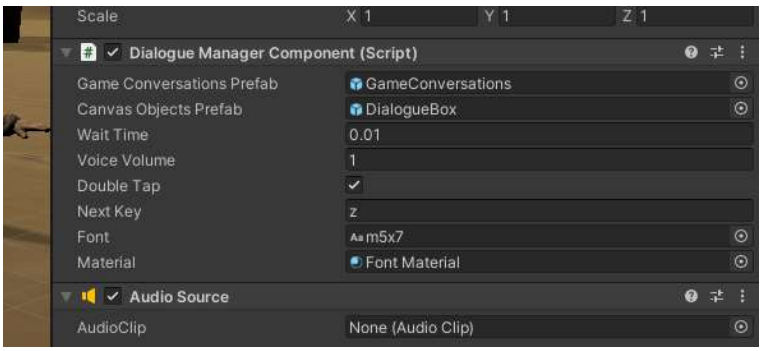




Una vez tengamos nuestro personaje, en la jerarquía de objetos de la escena añadimos al objeto con el que vayamos a interactuar un Box Collider y un GameConversationsComponent, que es un script que controla las conversaciones. Este script se encuentra en la carpeta DialogueSystem>Scripts>GameComponents.



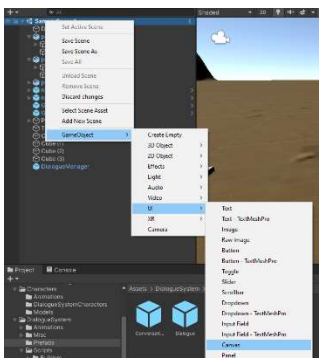
Ahora, añadimos un Prefab fundamental a la jerarquía de la escena: DialogueManager, que contiene los scripts encargados de controlar todo el sistema de diálogos de todos los personajes. Viene configurado de antemano, pero podemos cambiar alguno de los parámetros de su script asociado, DialogueManagerComponent, como por ejemplo Next Key, que indica la tecla que hará progresar los diálogos.

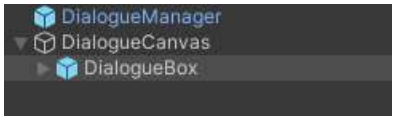


Aquí ya es posible que tengamos un problema: si estamos seguros de que vamos a contar con un teclado en los sistemas para los que diseñemos la aplicación, no hay problema, pero si vamos a hacer una build para móviles, por ejemplo, y especialmente en un entorno VR, puede ser una buena idea ahondar en el código y hacer que los diálogos puedan pasarse automáticamente, o cualquier otro modo que se nos ocurra y que no nos haga depender de teclado. Para más información de cómo está implementado el código y de dónde habría que tocarlo, revisar los Apéndices.

Para el canvas donde se mostrará la interfaz de las cajas de diálogo, creamos un UI-Canvas en la jerarquía de los objetos de la escena, a la que llamaremos DialogueCanvas. Solo necesitaremos un canvas de este tipo.

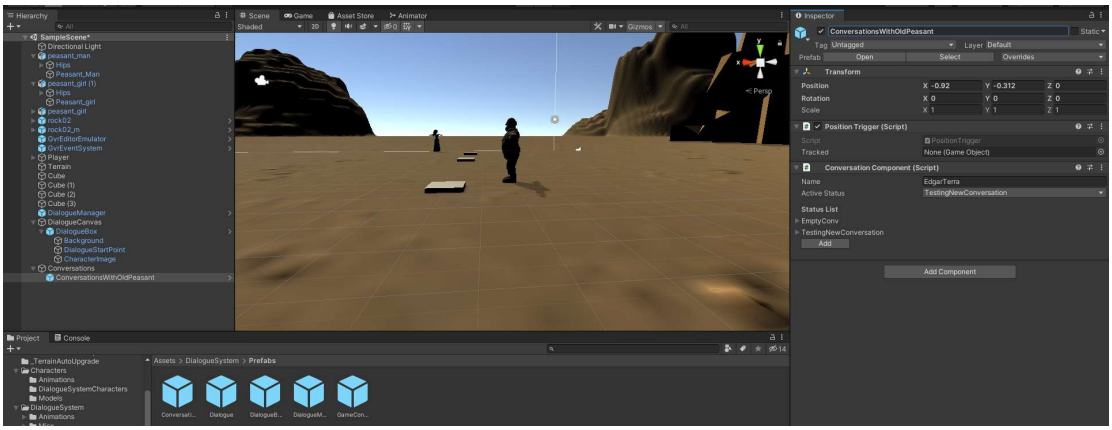
Dentro de DialogueSystem>Prefabs hay un Prefab llamado DialogueBox, que corresponde a la caja de diálogo que viene por defecto. La arrastramos dentro del DialogueCanvas que acabamos de crear, no siendo necesario tocar nada más si nos conformamos con el diseño base de la caja de diálogos.



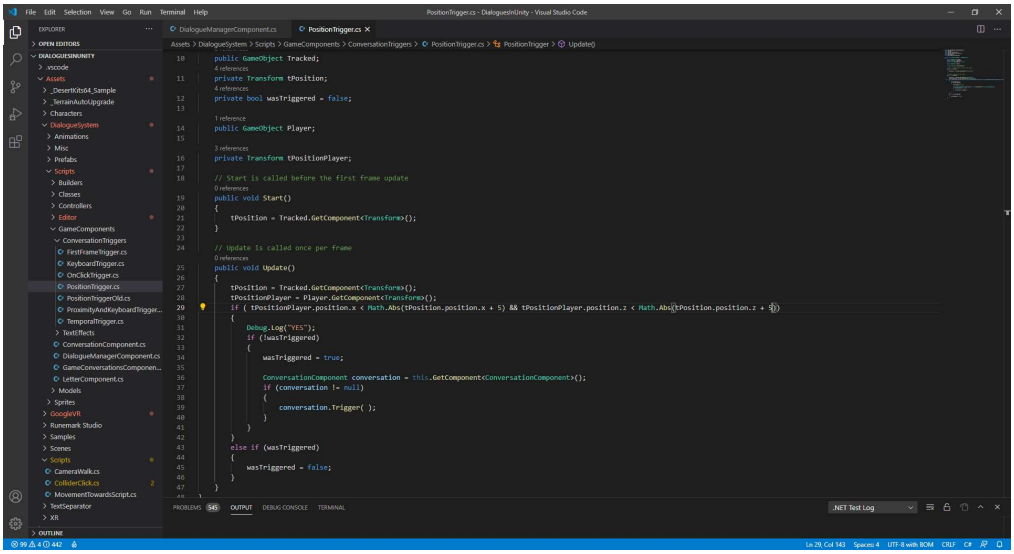


El último paso es crear las propias conversaciones entre los personajes, que es un proceso tan simple como todos los anteriores. Creamos un nuevo objeto vacío en la jerarquía al que llamaremos

Conversations. Dentro de éste, arrastraremos prefabs de tipo “Conversation”, tantos como distintas conversaciones vayamos a querer.

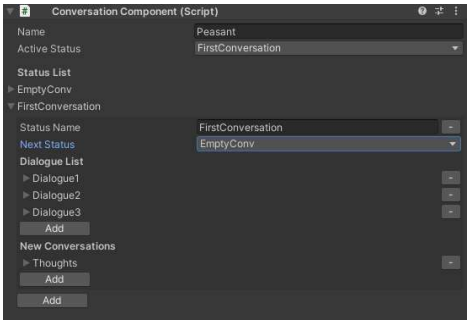


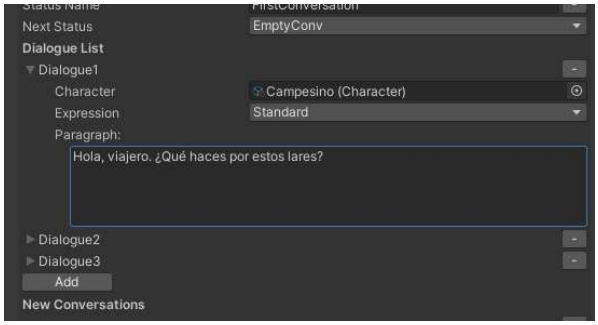
Estos objetos Conversation tienen un Trigger, que es el encargado de “disparar” la conversación cuando se den ciertas condiciones. Aunque viene uno por defecto, avisamos de que su implementación no es la mejor, y es recomendable crear Triggers nuevos que se ajusten a nuestras necesidades para cada situación. Un ejemplo de un Trigger correctamente implementado que dispara la conversación cuando el jugador se encuentra a cierta distancia del otro objeto sería como este:



```
public GameObject Tracked;
private Transform tPosition;
private bool wasTriggered = false;

public GameObject Player;

private Transform tPositionPlayer;

// Start is called before the first frame update
public void Start()
{
    tPosition = Tracked.GetComponent<Transform>();
}

// Update is called once per frame
public void Update()
{
    tPosition = Tracked.GetComponent<Transform>();
    tPositionPlayer = Player.GetComponent<Transform>();
    if ( tPositionPlayer.position.x < Math.Abs(tPosition.position.x + 5) && tPositionPlayer.position.z < Math.Abs(tPosition.position.z + 5))
    {
        Debug.Log("YES");
        if (!wasTriggered)
        {
            wasTriggered = true;

            ConversationComponent conversation = this.GetComponent<ConversationComponent>();
            if (conversation != null)
            {
                conversation.Trigger( );
            }
        }
    }
    else if (wasTriggered)
    {
        wasTriggered = false;
    }
}
```

Por otro lado, cada Conversation tiene un ConversationComponent, que es donde vamos a poder configurar la conversación, incluyendo nombre, el active status (desde dónde empieza la conversación por defecto) y una lista de status. Cada status correspondería a una conversación distinta, y le podemos ir añadiendo tantas líneas de diálogo como queramos. A su vez, cada línea de diálogo tiene asociado un personaje de los que hemos creado y una de sus expresiones, lo que hará que en el momento de mostrar dicha línea de diálogo se muestre la foto correspondiente a dicha expresión. Esto nos permite crear conversaciones en las que interviene más de un personaje. Además, en el momento en que terminemos una conversación/status, la siguiente vez que interactuemos con el objeto pasaremos a la conversación asociada en Next Status.





Un par de consideraciones a tener en cuenta:

1) La funcionalidad no parece estar correctamente implementada para el cambio de un status a otro, no se registra el Next Status y por tanto por más que interactuemos, solo se mostrará la primera conversación en la primera de las interacciones
2) Si no vamos a tocar nada del diseño de la DialogueBox, tenemos que tener cuidado con la longitud de cada línea de diálogo, porque es muy fácil que el texto se nos desborde

Con estos pasos ya tendríamos todo lo necesario para poder tener un primer sistema de diálogos en nuestro juego.

PROS Y CONTRAS

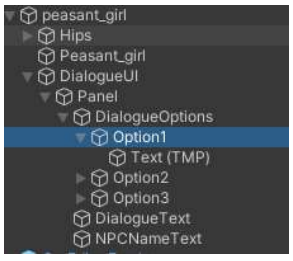
La gran ventaja de este sistema de diálogos es que ya está creado, simplemente tenemos que incluirlo en el proyecto que estemos desarrollando. Es relativamente sencillo de incorporar (siempre y cuando uno sepa qué tiene que hacer) y visualmente encaja muy bien con juegos de estética retro o RPGs, en los que los diálogos son un elemento clave. Sin embargo, tiene muchos problemas: la documentación es inexistente, no es sencillo modificar el código salvo en ciertos sitios porque tiene muchísimos scripts relacionados entre sí, la adaptación a VR funciona bastante mal y, por último, hay funcionalidades que no están correctamente implementadas y hacen que el sistema no tenga el comportamiento esperado. Como se ha podido ver, tampoco permite configurar diálogos con múltiple respuesta, que puede ser muy interesante en ciertos juegos. Además, tal como está depende de que el usuario tenga un teclado, lo que no lo hace ideal para juegos de móvil.

SEGUNDA PARTE: SISTEMA DE DIÁLOGOS "DESDE CERO"

En esta segunda parte hemos seguido un tutorial que permite crear un sistema de diálogos de múltiple opción e integrarlo en VR. El tutorial está disponible en el siguiente enlace: https://www.youtube.com/watch?v=wBFsA7rzNMQ&ab_channel=JorgeBlanco. Asimismo, se han consultado otros tutoriales de creación de sistemas de diálogos, pero pocos están específicamente orientados a VR.

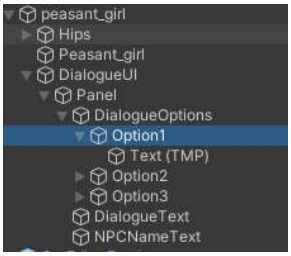
MODELO DEL SISTEMA DE DIÁLOGOS

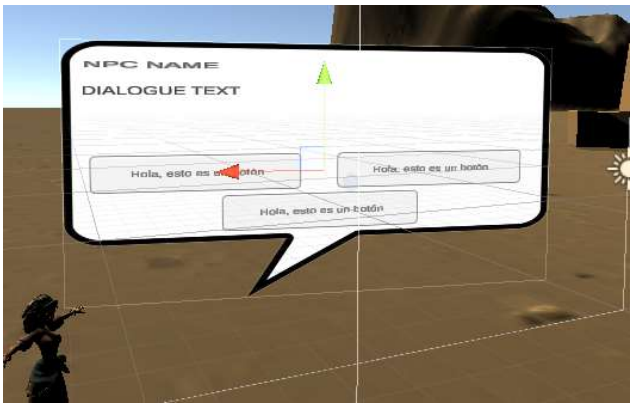
Cabe destacar que, en este sistema, cada personaje u objeto con el que interactuemos será quien encapsule el manejo de su propio diálogo, lo que incluye la interfaz donde se despliegan las líneas de diálogo y las opciones, siendo la jerarquía la siguiente:



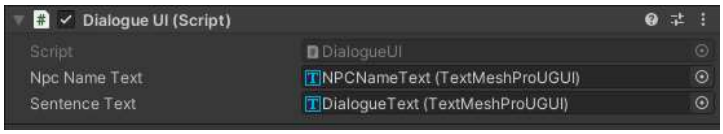
INTERFAZ DE DIÁLOGOS

En ella incluiremos dos campos de texto, uno que desplegará las líneas de diálogo y otro para el nombre del personaje que esté hablando. Asimismo, incluimos una serie de botones, tantos como queramos, que corresponderán a las opciones que podremos elegir. A la hora de que se desarrolle una conversación, solo se desplegarán en pantalla tantos botones como opciones haya para elegir en ese punto del diálogo; esto va controlado por código.





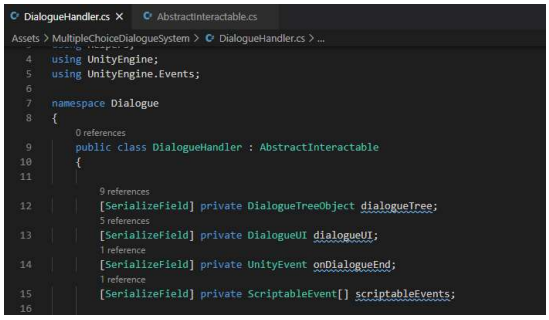
Respecto a la DialogueUI, solo queda que le añadamos el script DialogueUI, que se encargará de gestionar cómo aparecen los botones en pantalla, el texto que se despliega, etc.:



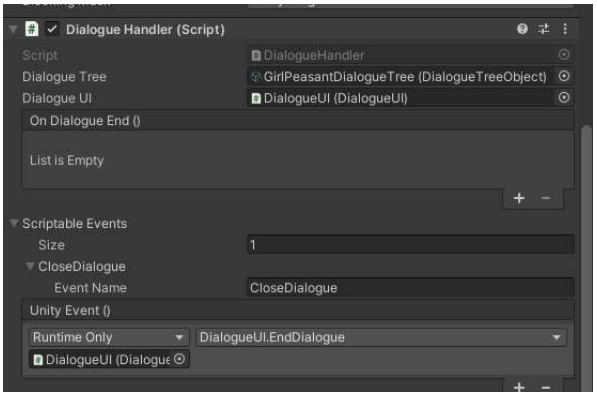
DIALOGUE HANDLER

DialogueHandler es el script que controla el flujo de un determinado diálogo. Contiene una serie de campos que debemos editar desde el inspector, entre ellos el árbol de diálogo correspondiente, que es la representación de una conversación de múltiple opción. En el tutorial seguido, esta clase DialogueHandler hereda de MonoBehaviour, pero por conveniencia, nosotros hemos hecho que herede de AbstractInteractable, porque si no en nuestro proyecto no se registra el evento OnPointerClick, que hace que la ventana de diálogo se despliegue. Para más información de cómo está implementado AbstractInteractable, ver los apéndices.

En cuanto a DialogueHandler, contiene un método que registra la interacción con el puntero y despliega el diálogo, así como un método ContinueDialogue que detecta en qué punto del diálogo nos encontramos y delega el manejo de esa siguiente unidad de diálogo en una función específica. Asimismo, podemos configurar la acción que queremos que se realice en el momento de que el diálogo finalice.



A aquel objeto con el que vayamos a interactuar hemos de asignarle un DialogueHandler y pasarle a éste el árbol de diálogo (que veremos más adelante) correspondiente a la conversación que vamos a mantener:



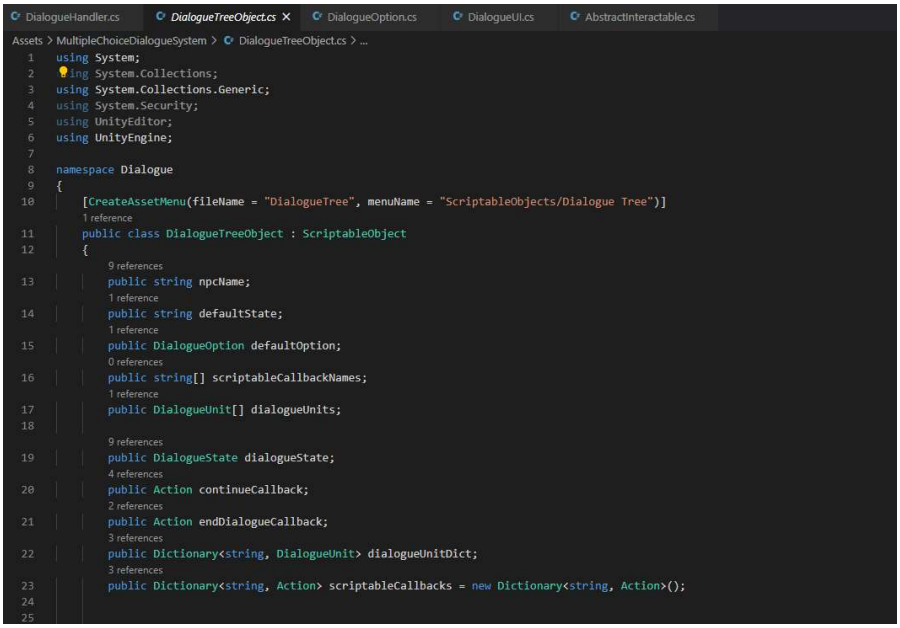
OPCIONES DE DIÁLOGO

En la jerarquía de la escena, las opciones de diálogo son botones con los que podemos interactuar mediante un puntero en VR, para lo que debemos asociarles un GvrPointerGraphicRaycaster. Podemos crear tantos como queramos, ya que en cada momento solo se desplegarán tantos como opciones hayamos configurado para ese determinado punto de la conversación.

En cuanto al script de las DialogueOptions, constan de un texto, que será el que se despliegue en el botón con el que interactuemos, y un evento, que se corresponde a la acción que queremos que se ejecute al interactuar con la opción. Veremos en el apartado siguiente cómo se configuran las opciones desde el inspector.

CREACIÓN DE ÁRBOLES DE DIÁLOGO (DIALOGUE TREES)

Los árboles de diálogo representan las conversaciones y los distintos caminos que podemos tomar en cada una de ellas, y son posiblemente la parte más compleja de este sistema.



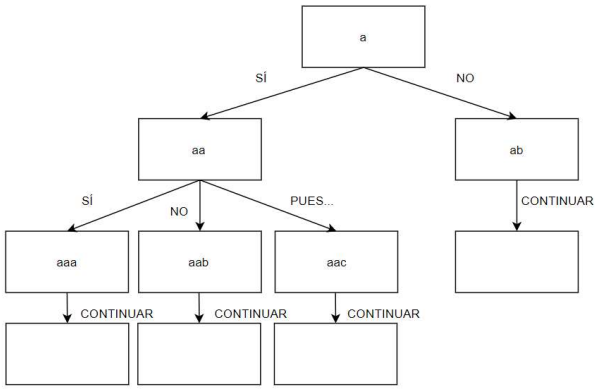
Lo primero de todo, el hecho de que sea un ScriptableObject va a permitirnos almacenar gran cantidad de datos en cada uno de los árboles. Como podemos ver, consta, entre otros, de:

- Nombre del personaje con el que mantenemos la conversación (npcName)
- El estado por defecto en el que se encuentra cuando se interactúa por primera vez con él (defaultState)
- Una opción que podemos configurar para que salga por defecto cuando nosotros no damos ninguna (defaultOption)

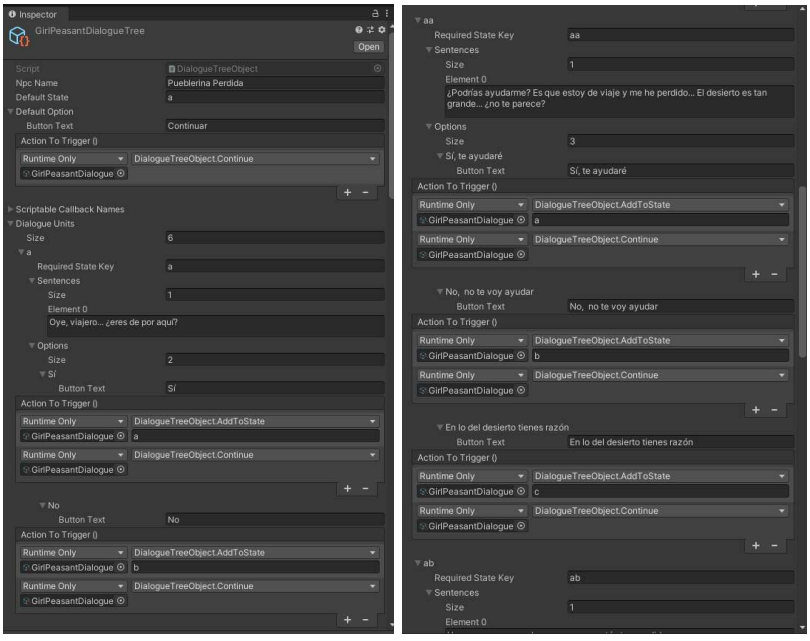
- Una lista de DialogueUnits, es decir, de las piezas que conforman la conversación (dialogueUnits). Cada DialogueUnit consta a su vez de una lista de sentencias (líneas de diálogo) y de una lista de opciones de diálogo asociadas
- Una lista de strings cuyos valores corresponden a los nombres de funciones callback (scriptableCallbackNames)

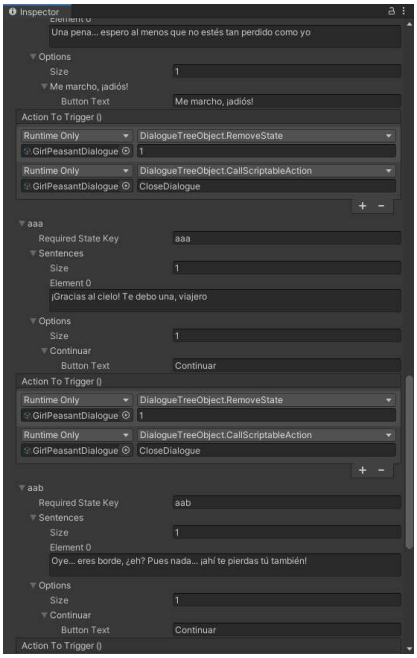
Para configurar una conversación solo tenemos que, a partir de este DialogueTreeObject, crear un nuevo asset desde Assets>Create>ScriptableObject>DialogueTree. A partir de ese punto, podremos modificar todos los valores mencionados en el párrafo anterior, y crear conversaciones completas con múltiples caminos. Ojo: cada conversación necesitará de su propio objeto DialogueTree.

Para esta parte es recomendable hacer previamente un esquema de cómo queremos que se desarrolle la conversación, puesto que puede ser un poco complicado ir creando el diálogo directamente en el editor sin planificación. Pensemos en una conversación como un árbol en el que en cada nodo nos encontremos en un determinado estado, y el hecho de escoger una rama u otra determine el siguiente estado; de esta manera podremos ser capaces de trazar el recorrido de la conversación en cualquier momento. Un modelo de esquema podría ser el siguiente, en donde cada nodo incluye el estado en el que queda tras llegar a él y las flechas que unen los nodos representan las opciones elegidas para pasar de un nodo a otro.



Esto, trasladado al inspector, que será desde donde modifiquemos los diálogos, sería como sigue:





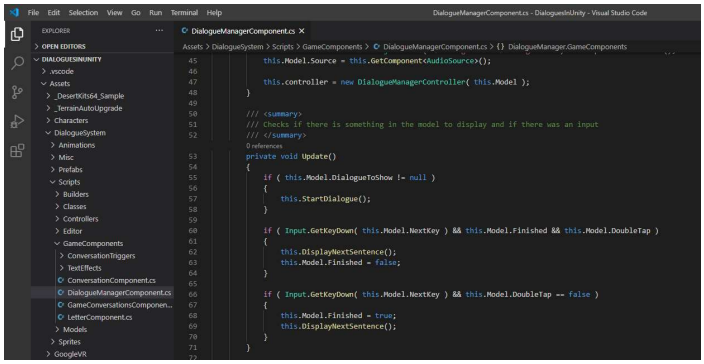
PROS Y CONTRAS

La principal ventaja de este sistema es que, en principio, está preparado específicamente para VR y no tendríamos problemas en integrarlo en nuestro proyecto. Además, al ir implementándolo desde cero, podemos ver cómo interactúan todos los componentes entre sí y, lo más importante, permite tener diálogos de tipo branching. El único problema que presenta para nosotros es que el diseño de las conversaciones es un poco complicado, ya que uno tiene que tener muy claro cómo se van desarrollando y a qué estado pasamos dada una determinada opción.

APÉNDICES

MODIFICACIÓN DEL CÓDIGO DE DIALOGUESYSTEM - INTEGRACIÓN CON TECLADO

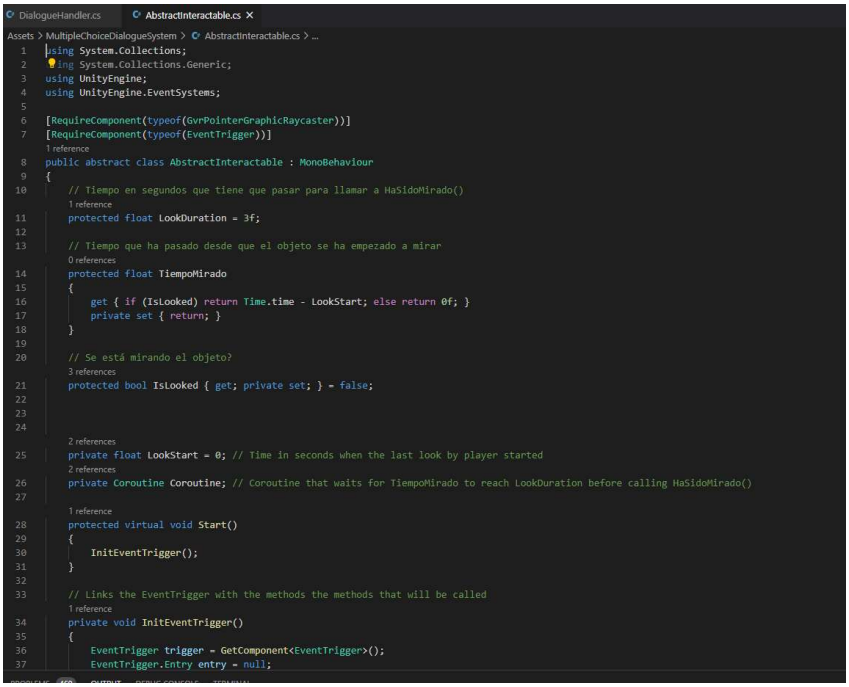
En el script DialogueManagerComponent se comprueba si se recibe un input por parte del usuario, y en caso afirmativo compara la tecla del input con la asociada a Next Key. El problema: dependencia total de que dispongamos de teclado.



Una posible solución es deshacernos del reconocimiento de input en las condiciones, lo que hará que el diálogo se vaya desplegando automáticamente. En ese caso solo habría que ajustar la velocidad a la que se van pasando las líneas de diálogo y el tiempo de espera entre que se termine de desplegar una línea y la siguiente. Otra opción sería comprobar el tipo de input disponible y ajustarnos en cada caso, aunque eso probablemente conllevaría tocar bastante más código, pero sería una solución más elegante y cómoda para el usuario en muchos casos.

ABSTRACT INTERACTABLE

Esta clase ha sido empleada en el proyecto de nuestro grupo de prácticas, el juego de Harry Potter, para poder interactuar con los objetos del entorno en VR; hemos decidido reutilizarla aquí porque supone la diferencia entre que el DialogueHandler funcione correctamente o no. El código es el siguiente:



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

[RequireComponent(typeof(GvrPointerGraphicRaycaster))]
[RequireComponent(typeof(EventTrigger))]
public abstract class AbstractInteractable : MonoBehaviour
{
    // Tiempo en segundos que tiene que pasar para llamar a HaSidoMirado()
    protected float LookDuration = 3f;

    // Tiempo que ha pasado desde que el objeto se ha empezado a mirar
    protected float TiempoMirado
    {
        get { if (IsLooked) return Time.time - LookStart; else return 0f; }
        private set { return; }
    }

    // Se está mirando el objeto?
    protected bool IsLooked { get; private set; } = false;



    private float LookStart = 0; // Time in seconds when the last look by player started
    private Coroutine Coroutine; // Coroutine that waits for TiempoMirado to reach LookDuration before calling HaSidoMirado()

    protected virtual void Start()
    {
        InitEventTrigger();
    }

    // Links the EventTrigger with the methods the methods that will be called
    private void InitEventTrigger()
    {
        EventTrigger trigger = GetComponent<EventTrigger>();
        EventTrigger.Entry entry = null;
```

```
    // Links the EventTrigger with the methods the methods that will be called
    private void InitEventTrigger()
    {
        EventTrigger trigger = GetComponent<EventTrigger>();
        EventTrigger.Entry entry = null;

        // Add click
        entry = new EventTrigger.Entry();
        entry.eventID = EventTriggerType.PointerDown;
        entry.callback.AddListener((data) => { OnPointerClick(); });
        trigger.triggers.Add(entry);

        // Add pointer enter
        entry = new EventTrigger.Entry();
        entry.eventID = EventTriggerType.PointerEnter;
        entry.callback.AddListener(
            (data) => {
                this.IsLooked = true;
                LookStart = Time.time;
                Coroutine = StartCoroutine(WaitCoroutine());
                OnPointerEnter();
            }
        );
        trigger.triggers.Add(entry);

        // Add pointer exit
        entry = new EventTrigger.Entry();
        entry.eventID = EventTriggerType.PointerExit;
        entry.callback.AddListener(
            (data) => {
                this.IsLooked = false;
                StopCoroutine(Coroutine);
                OnPointerExit();
            }
        );
        trigger.triggers.Add(entry);
    }

    // Espera el tiempo necesario antes de llamar a HaSidoMirado
    // Si el jugador deja de mirar, la coroutine se detiene con StopCoroutine
    private IEnumerator WaitCoroutine()
    {
        yield return new WaitForSeconds(LookDuration);
```

```
        trigger.triggers.Add(entry);

        // Add pointer exit
        entry = new EventTrigger.Entry();
        entry.eventID = EventTriggerType.PointerExit;
        entry.callback.AddListener(
            (data) => {
                this.IsLooked = false;
                StopCoroutine(Coroutine);
                OnPointerExit();
            }
        );
        trigger.triggers.Add(entry);
    }

    // Espera el tiempo necesario antes de llamar a HaSidoMirado
    // Si el jugador deja de mirar, la coroutine se detiene con StopCoroutine
    private IEnumerator WaitCoroutine()
    {
        yield return new WaitForSeconds(LookDuration);
        HaSidoMirado();
    }

    // Llamado cuando el jugador hace click
    public virtual void OnPointerClick() { }

    // Llamado cuando el jugador empieza a mirar el objeto
    public virtual void OnPointerEnter() { }

    // Llamado cuando el jugador deja de mirar el objeto
    public virtual void OnPointerExit() { }

    // Llamado cuando el objeto ha sido mirado durante LookDuration segundos
    public virtual void HaSidoMirado() { }
}
```