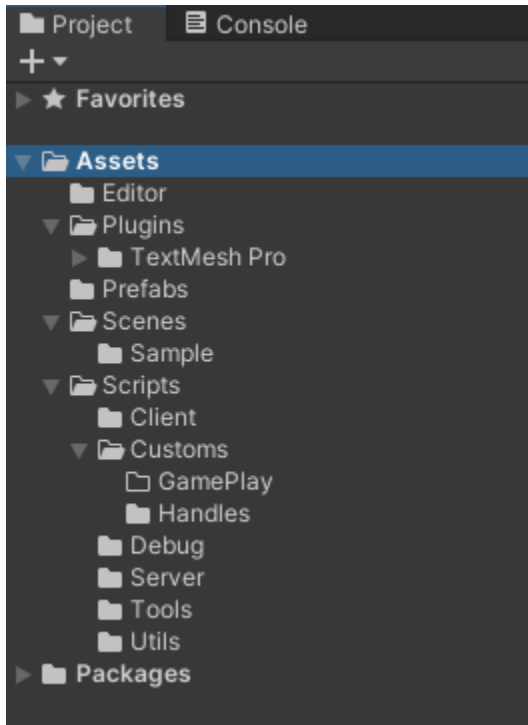


Manual Woo

Woo es una estructura diseñada para facilitar la conexión entre un ordenador y un dispositivo móvil (Android).

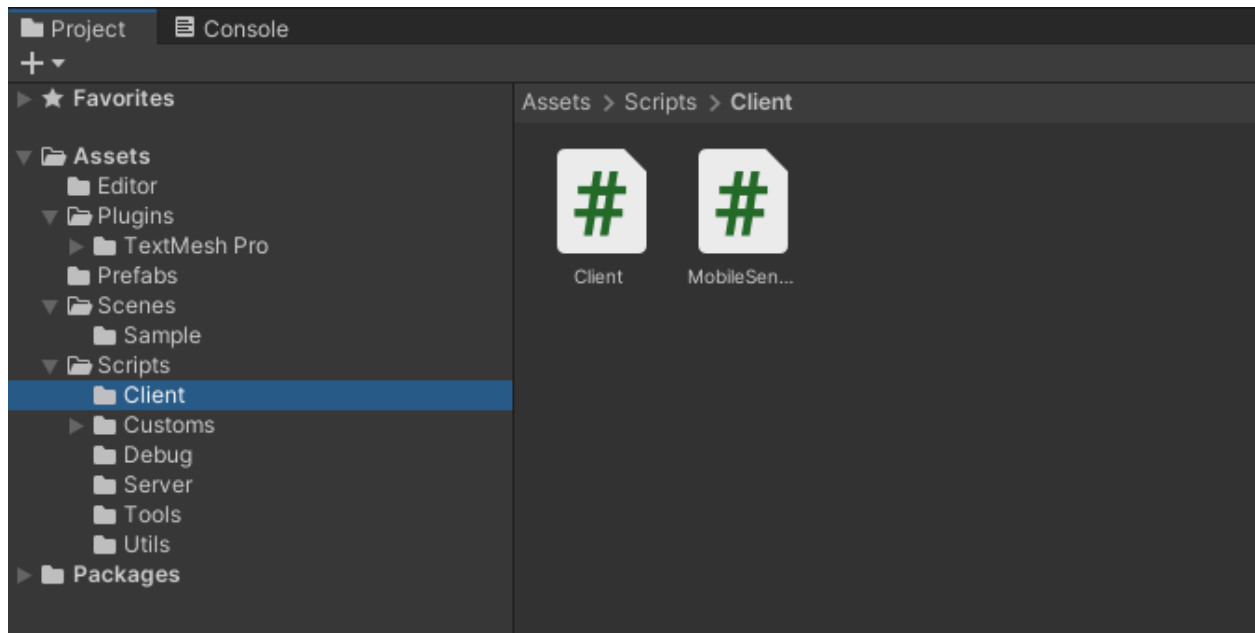
Arquitectura:



Dentro de la carpeta Scene/Sample, se encuentran varias escenas de demostración para probar la funcionalidad del proyecto.

Dentro de la carpeta Scripts, están todos los scripts utilizados en el proyecto. Los scripts creados por el usuario deben colocarse dentro de Scripts/Customs.

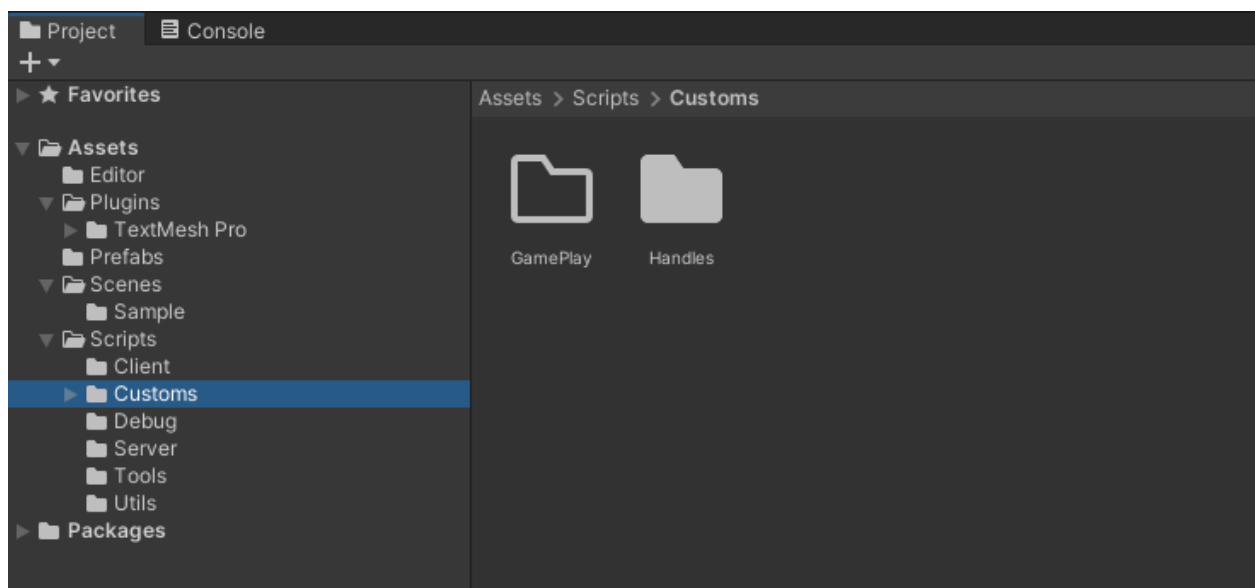
Client:



En la carpeta Client, podemos encontrar dos scripts. El más importante es Client, donde se gestiona toda la lógica de interacción con el servidor. Contiene una clase singleton Client que envía y recibe mensajes al servidor.

Además, encontramos el script MobileSensor, que recopila información de los sensores del móvil, como aceleraciones, velocidad, gravedad, etc.

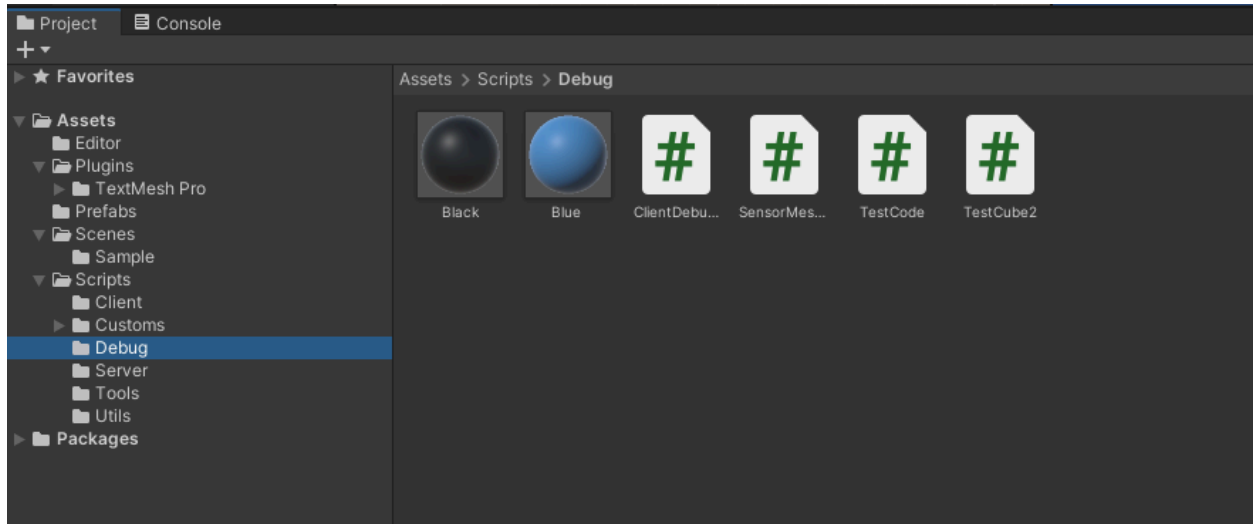
Customs:



Esta es la carpeta donde el usuario debe colocar sus propios scripts. En la carpeta gameplay, se ubicarán todos los scripts que controlan el flujo del juego. Por otro lado, en la carpeta Handles, se encontrarán todos los manejadores que interpretan la información recibida, tanto en el cliente como en el servidor.

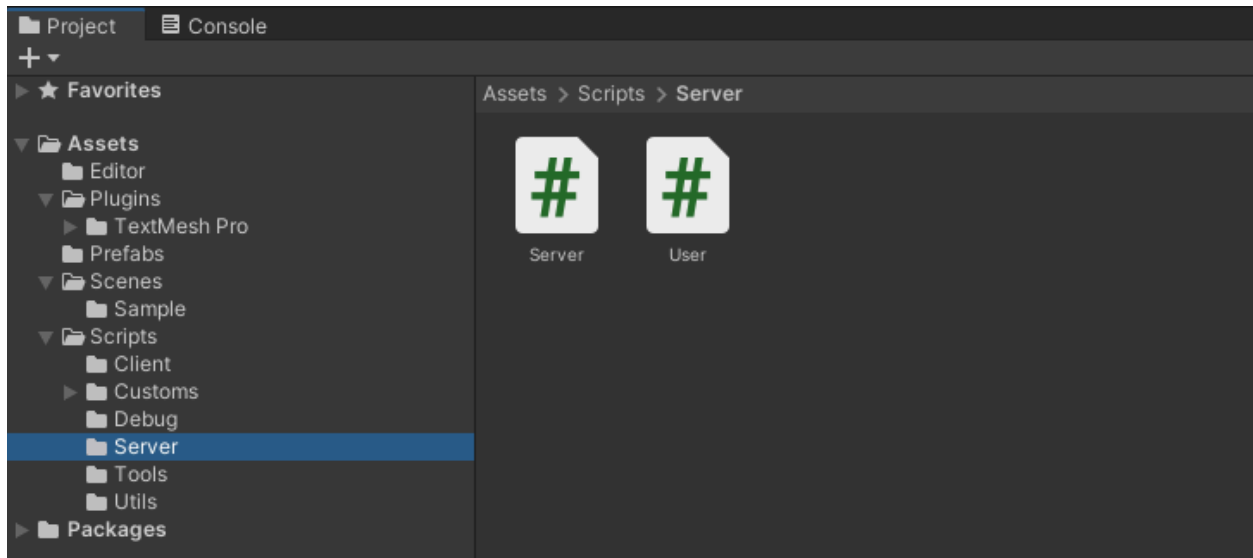
En la carpeta Handles, se incluyen dos scripts de ejemplo: uno para gestionar los mensajes recibidos en el cliente y otro en el servidor, que se explicará más adelante.

Debug:



En la carpeta Debug, se encuentran scripts utilizados durante el desarrollo y las demostraciones. Estos scripts no deben incluirse en la versión final del juego.

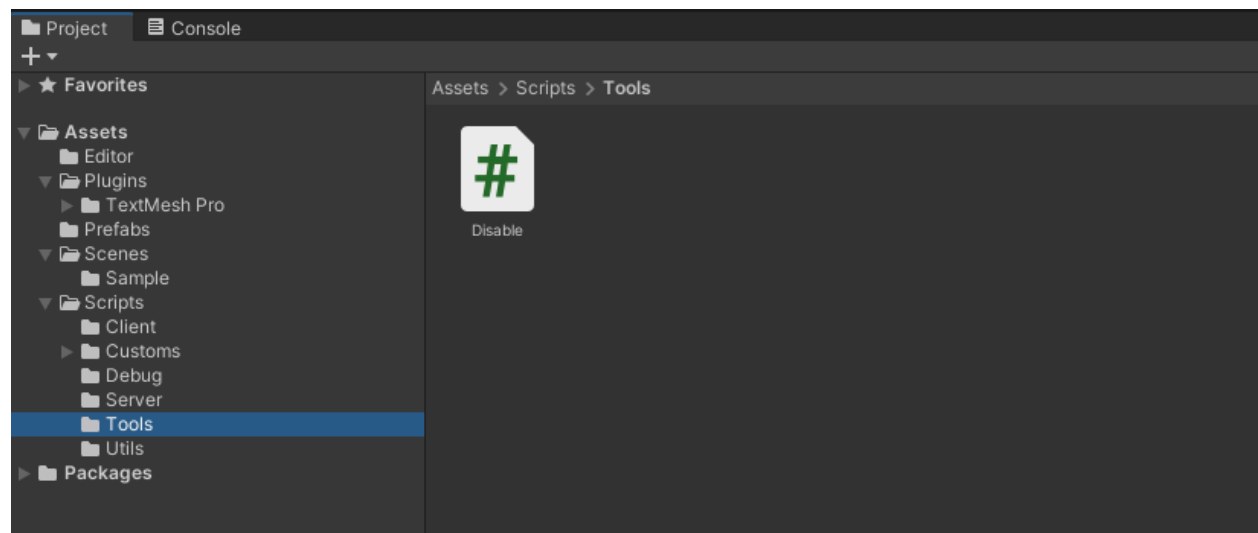
Server:



En la carpeta Server, se encuentran los scripts utilizados en el servidor. El script más importante es Server, que contiene una clase singleton Server donde se gestiona toda la información recibida de diferentes clientes.

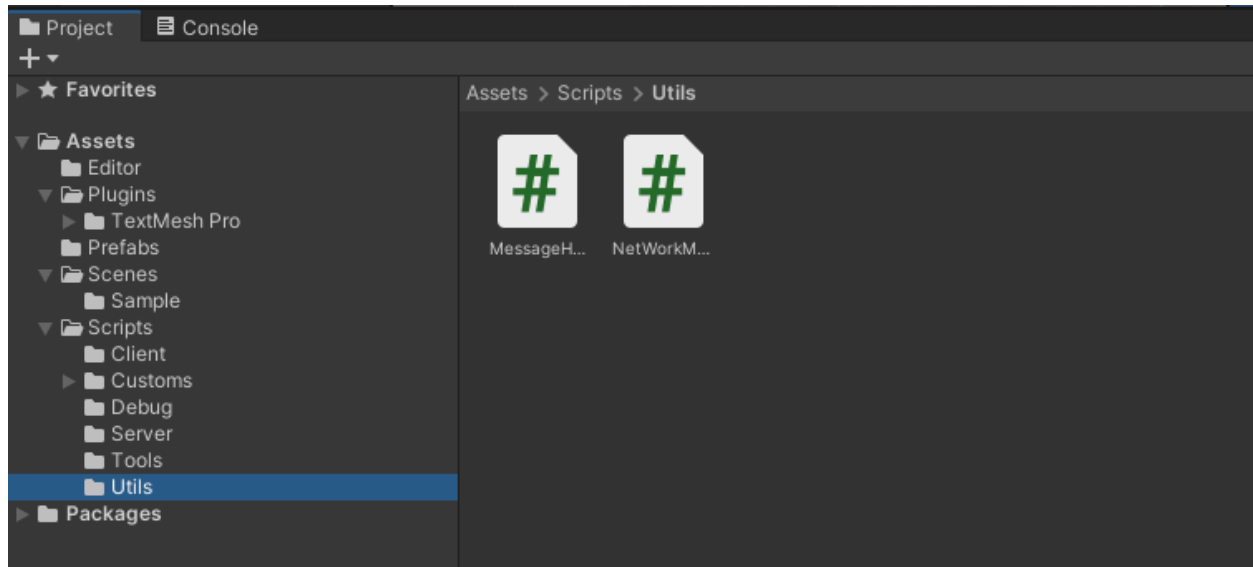
El script User contiene una clase de soporte para Server, utilizada para guardar los datos necesarios de un usuario. Puede ser personalizada para agregar más información si se desea.

Tools:



En la carpeta Tools, se encuentran scripts que no están directamente relacionados con el proyecto en sí, sino que se utilizan para personalizar el editor de Unity.

Utils:



En la carpeta Utils, se encuentran los mensajes utilizados para la interacción entre el servidor y el cliente. El script **MessageHandler** contiene una clase de soporte para **NetworkMessage** y no debe ser modificado por el usuario.

Por otro lado, el script **NetworkMessage** contiene todos los tipos de mensajes que pueden existir en el juego. Cuando se desea crear un nuevo mensaje, es necesario modificar varios lugares en este script.

Scenes

El juego debe contener al menos dos escenas: una para el móvil y otra para PC. Cada escena tendrá sus propios gameobjects. La clase Server debe existir únicamente en la escena de PC, mientras que la clase Client debe existir únicamente en la escena del móvil.

Server

El servidor es una clase singleton que gestiona las entradas de los clientes, los mensajes y también se encarga de enviar mensajes a los clientes (PC a Mobile).

Dentro de la clase, existe una función **StartServerAsync()**, que crea el servidor utilizando una dirección IP local y un puerto disponible. Esta función se llama en el método **Start()** de la clase, ya que se espera que el servidor se inicie inmediatamente al abrir el juego. Sin embargo, el usuario puede quitarla del método **Start()** y llamarla cuando lo desee para activar el servidor.

La clase incluye dos tipos de funciones para enviar mensajes: **SendMessageToClients()** y **SendMessageToClient()**. La primera se utiliza para enviar un mensaje a todos los clientes conectados, mientras que la segunda se utiliza para enviar un mensaje a un cliente específico.

Finalmente, la función **RegisterMessageEvent(NetworkMessageType type, Action<NetworkMessage> callbackFunc)** se utiliza para registrar un evento que se activará cuando se reciba un mensaje de un tipo específico.

Client

El cliente es una clase singleton que envía y recibe mensajes al servidor (Mobile a PC).

En la clase Client, existe una función llamada **RequestConnectToServer()** que utiliza la dirección IP y el puerto almacenados para conectarse al servidor. Esta función debe ser llamada por el usuario cuando desee establecer la conexión.

También existen las funciones **SetServerPort(int port)** y **SetServerIP(string ip)** para configurar el puerto y la dirección IP del servidor.

La función **RegisterMessageEvent(NetworkMessageType type, Action<NetworkMessage> callbackFunc)** funciona de manera similar a la del servidor, pero en este caso se utiliza para el cliente.

NetMessage

Esta clase almacena todos los tipos de mensajes que existen en el juego. Cada vez que se desea agregar un nuevo tipo de mensaje, se deben seguir los siguientes pasos:

- 1 - Añadir un nuevo tipo de mensaje al **enum NetworkMessageType**.
- 2 - Crear una nueva clase que herede de **NetMessage** para el nuevo tipo de mensaje, siguiendo la estructura de otros mensajes.
- 3 - Dentro de la clase **NetworkPackage**, agregar una nueva acción al diccionario **_getDataActions**.
- 4 - Dentro de la clase **NetMessageFactory**, crear una función para inicializar una instancia del nuevo tipo de mensaje.

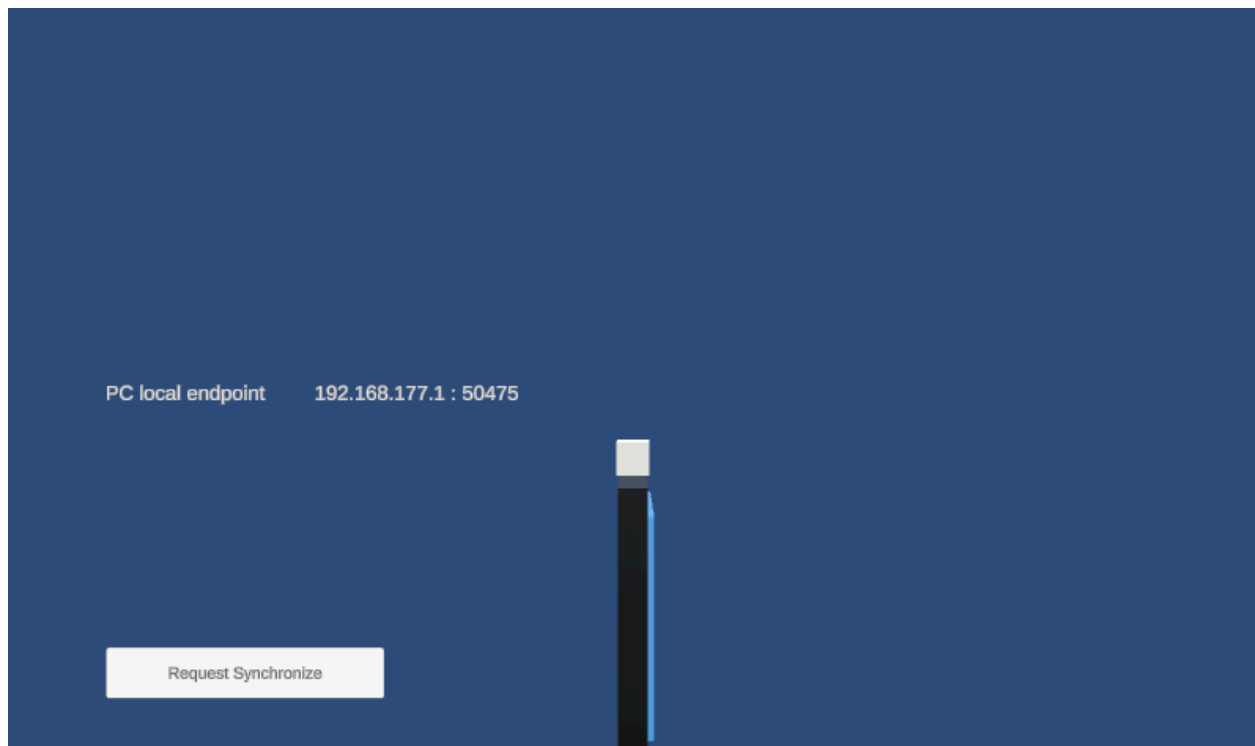
Demo

El proyecto incluye varias escenas de demostración ubicadas en la carpeta **Scene/Sample**. La escena **Mobile_Scene** debe compilarse para Android y la escena **PC_Scene** debe compilarse para Windows.

Al ejecutar la escena de PC, se iniciará automáticamente el servidor y se mostrará la dirección IP y el puerto que deben ser utilizados por los dispositivos móviles para conectarse.

Una vez que un usuario se conecta, puede hacer clic en el botón de sincronización de rotación para sincronizar el dispositivo móvil físico con el dispositivo móvil virtual.

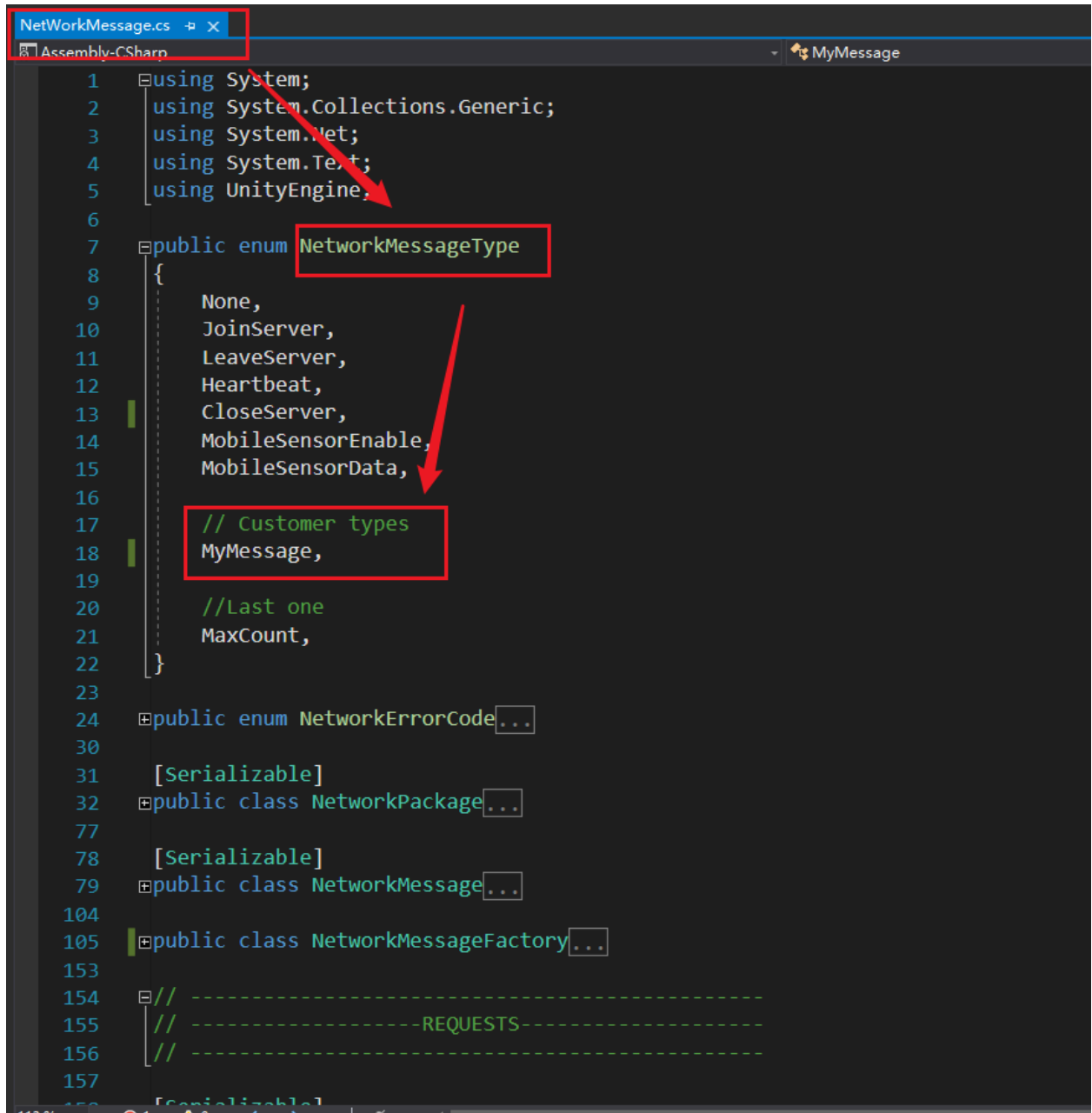
Los usuarios pueden crear sus propias escenas basándose en estas escenas de demostración.



Ejemplo de mensaje

El proyecto incluye scripts de ejemplo para crear un nuevo tipo de mensaje. Imaginemos que deseamos enviar un mensaje de texto al servidor desde un dispositivo móvil.

1. Agregar un nuevo tipo de mensaje al enum **NetworkMessageType**.



```
1 using System;
2 using System.Collections.Generic;
3 using System.Net;
4 using System.Text;
5 using UnityEngine;
6
7 public enum NetworkMessageType
8 {
9     None,
10    JoinServer,
11    LeaveServer,
12    Heartbeat,
13    CloseServer,
14    MobileSensorEnable,
15    MobileSensorData,
16
17    // Customer types
18    MyMessage,
19
20    //Last one
21    MaxCount,
22 }
23
24 public enum NetworkErrorCode...
30
31 [Serializable]
32 public class NetworkPackage...
77
78 [Serializable]
79 public class NetworkMessage...
104
105 public class NetworkMessageFactory...
153
154 // -----
155 // -----REQUESTS-----
156 // -----
157
158 [Serializable]
```

2. Crear una nueva clase que herede de **NetworkMessage** y agregar un campo de tipo string para almacenar el mensaje.


```

NetWorkMessage.cs
Assembly-CSharp
MyMessage

197 {
198     public MobileSensorData(uint uid, Dictionary<MobileSensorFlag, Vector3> mobileSensorData)
199     {
200         this.quat = quat;
201         rot = mobileSensorData[MobileSensorFlag.Rotation];
202         vel = mobileSensorData[MobileSensorFlag.Velocity];
203         acc = mobileSensorData[MobileSensorFlag.Acceleration];
204         grav = mobileSensorData[MobileSensorFlag.Gravity];
205     }
206
207     // 4 server
208     public Vector3 rot;
209     public Vector3 vel;
210     public Vector3 acc;
211     public Vector3 grav;
212     public Quaternion quat;
213 }
214
215 /// Server 2 Client
216 [Serializable]
217 public class MobileSensorEnable : NetworkMessage
218 {
219     // -----
220     // -----CUSTOM REQUESTS-----
221     // -----
222
223     public class MyMessage : NetworkMessage
224     {
225         public MyMessage(uint uid, string message) : base(NetworkMessageType.MyMessage, uid)
226         {
227             this.message = message;
228         }
229
230         public string message;
231     }
232 }

```

3. En la clase **NetworkPackage**, agregar una nueva acción para el nuevo tipo de mensaje.

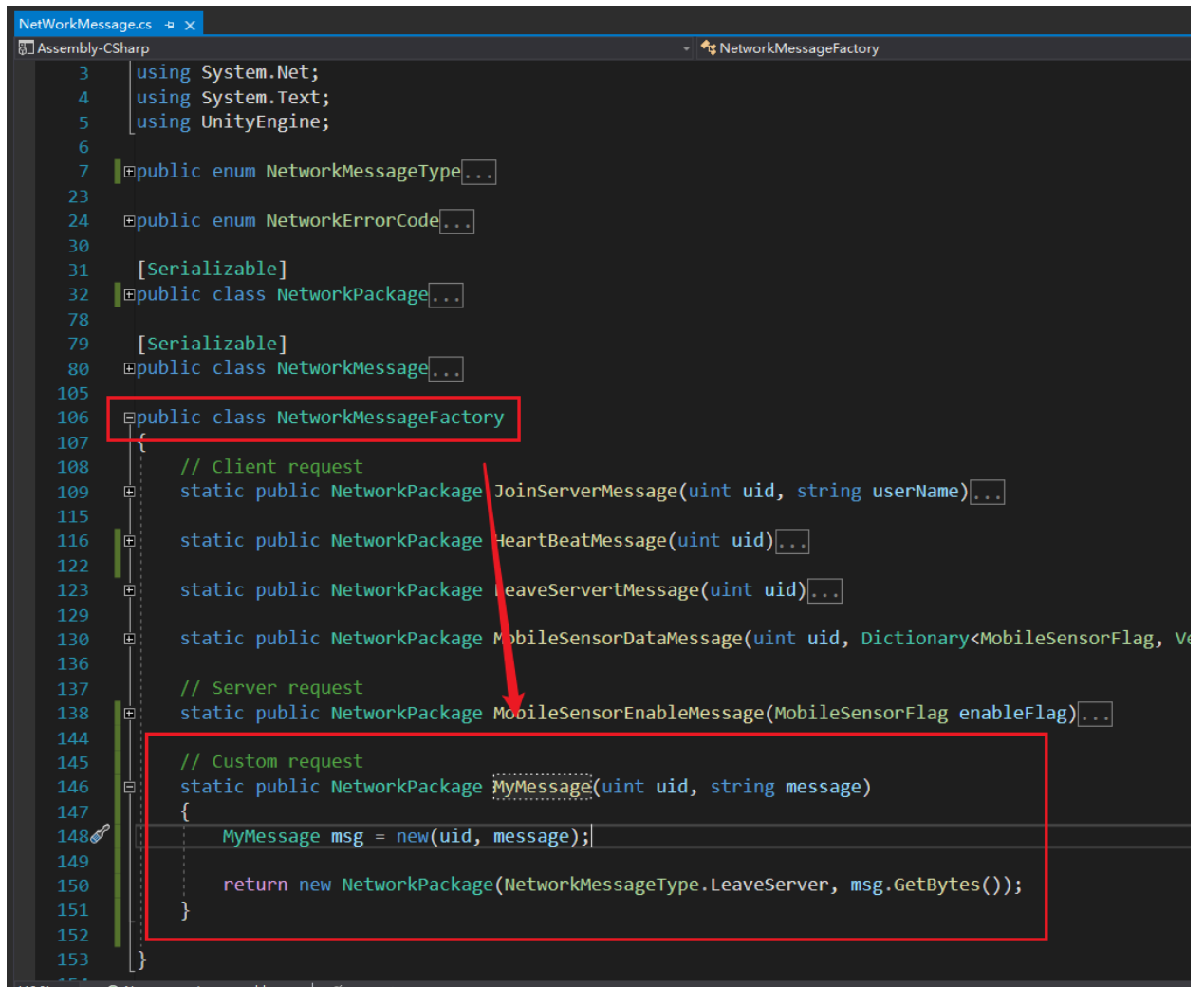
```

NetWorkMessage.cs
Assembly-CSharp
NetworkPackage

32 public class NetworkPackage
33 {
34     public NetworkPackage(NetworkMessageType type, byte[] data)
35     {
36         this.type = type;
37         this.data = data;
38     }
39
40     public byte[] GetBytes()
41     {
42         return data;
43     }
44
45     public NetworkMessage GetData()
46     {
47         return _getDataActions[type].Invoke(data);
48     }
49
50     public static NetworkMessage GetDataFromBytes(byte[] data, int lenght)
51     {
52         var jsonString = Encoding.ASCII.GetString(data, 0, lenght);
53         var networkPackage = JsonUtility.FromJson<NetworkPackage>(jsonString);
54         return networkPackage.GetData();
55     }
56
57     public NetworkMessageType type;
58     public byte[] data;
59
60     private static readonly Dictionary<NetworkMessageType, Func<byte[], NetworkMessage>> _getDataActions
61     {
62         {
63             NetworkMessageType.Heartbeat, GetData<HeartBeat> },
64             NetworkMessageType.JoinServer, GetData<JoinServer> },
65             NetworkMessageType.LeaveServer, GetData<LeaveServer> },
66             NetworkMessageType.MobileSensorEnable, GetData<MobileSensorEnable> },
67             NetworkMessageType.MobileSensorData, GetData<MobileSensorData> },
68             NetworkMessageType.MyMessage, GetData<MyMessage> },
69         }
70     };

```

4. En la clase **NetworkMessageFactory**, crear una función para inicializar una instancia del nuevo tipo de mensaje.



```
3 using System.Net;
4 using System.Text;
5 using UnityEngine;
6
7 public enum NetworkMessageType...
23
24 public enum NetworkErrorCode...
30
31 [Serializable]
32 public class NetworkPackage...
78
79 [Serializable]
80 public class NetworkMessage...
105
106 public class NetworkMessageFactory
107 {
108     // Client request
109     static public NetworkPackage JoinServerMessage(uint uid, string userName)...
115
116     static public NetworkPackage HeartBeatMessage(uint uid)...
122
123     static public NetworkPackage LeaveServerMessage(uint uid)...
129
130     static public NetworkPackage MobileSensorDataMessage(uint uid, Dictionary<MobileSensorFlag, V...
136
137     // Server request
138     static public NetworkPackage MobileSensorEnableMessage(MobileSensorFlag enableFlag)...
144
145     // Custom request
146     static public NetworkPackage MyMessage(uint uid, string message)
147     {
148         MyMessage msg = new(uid, message);
149
150         return new NetworkPackage(NetworkMessageType.LeaveServer, msg.GetBytes());
151     }
152
153 }
```

5. En la carpeta **Scripts/Custom/Handles**, se agregarán dos scripts: **HandleMyMsgClient** y **HandleMyMsgServer**.

En la carpeta **Scripts/Custom/Handles**, se agregarán dos scripts: **HandleMyMsgClient** y **HandleMyMsgServer**.

En la clase **HandleMyMsgServer**, se creará una función **OnReceiveMessage** que se activará cuando se reciba un mensaje del tipo especificado. En esta función, se puede implementar la lógica que se ejecutará al recibir el mensaje.

```
HandleMyMsgServer.cs x HandleMyMsgClient.cs
Assembly-CSharp HandleMyMsgServer

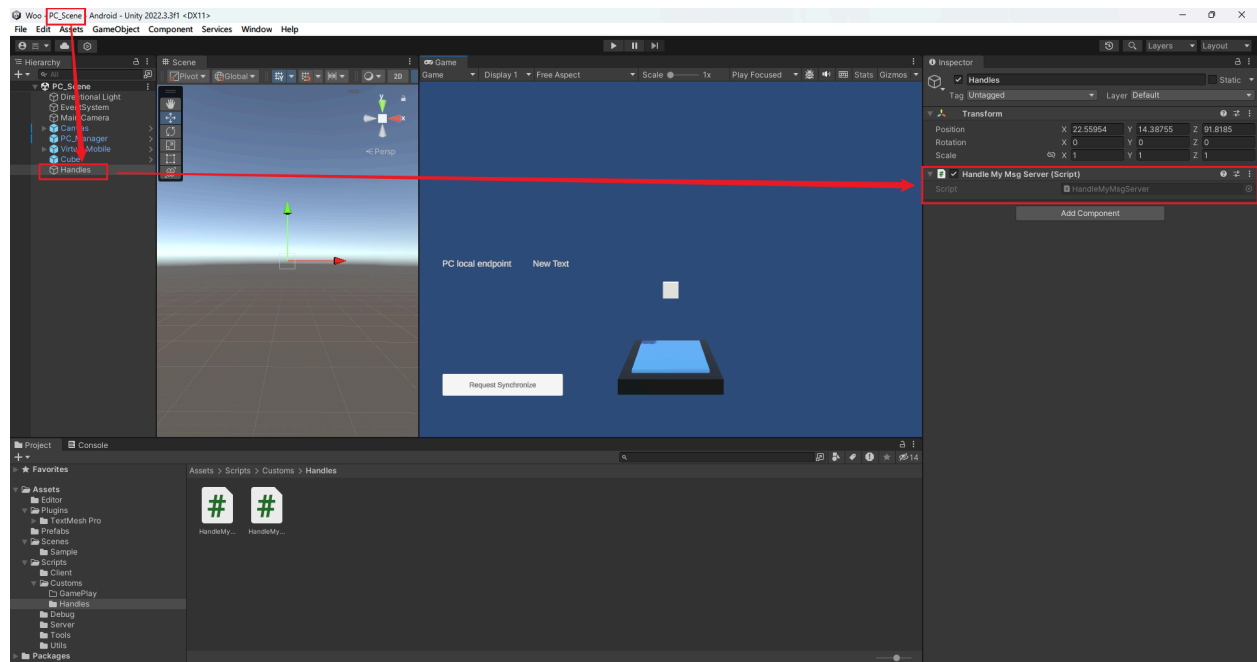
1 using UnityEngine;
2
3 public class HandleMyMsgServer : MonoBehaviour
4 {
5     // Start is called before the first frame update
6     void Start()
7     {
8         Server.instance.RegisterHandleEvent(NetworkMessageType.MyMessage, OnReveiveMessage);
9     }
10
11 private void OnReveiveMessage(NetworkMessage data)
12 {
13     var message = data as MyMessage;
14
15     Debug.Log($"Message receive from user {message.ownerUID}: {message.message}");
16
17     message.successful = true;
18
19     Server.instance.SendMessageToClient(message);
20 }
21
22 private void OnDestroy()
23 {
24     Server.instance.UnregistHandleEvent(NetworkMessageType.MyMessage);
25 }
26 }
27
```

En la clase **HandleMyMsgClient**, se creará una función similar para manejar los mensajes recibidos en el cliente.

```
Archivo Editar Ver Git Proyecto Compilar Depurar Prueba Analizar Herramientas Extensiones Ventana Ayuda Buscar (Ctrl+Q)
Debug Any CPU Asociar a Unity
HandleMyMsgServer.cs HandleMyMsgClient.cs x
Assembly-CSharp HandleMyMsgClient

1 using UnityEngine;
2
3 public class HandleMyMsgClient : MonoBehaviour
4 {
5     // Start is called before the first frame update
6     void Start()
7     {
8         Client.instance.RegisterHandleEvent(NetworkMessageType.MyMessage, OnReveiveMessage);
9     }
10
11 private void OnReveiveMessage(NetworkMessage data)
12 {
13     var message = data as MyMessage;
14
15     Debug.Log($"Server receive message with succceful {message.successful}");
16 }
17
18 private void OnDestroy()
19 {
20     Client.instance.UnregistHandleEvent(NetworkMessageType.MyMessage);
21 }
22 }
23
```

Una vez creados los scripts de handle, se debería colocarlo en el GameObject que toca tanto en la escena de PC como en la escena de móvil.



Exportar el juego

Al exportar el juego, se debe realizar una exportación por cada plataforma y configurar el tamaño de la ventana según corresponda.

