Game Synchronization Developer Guide

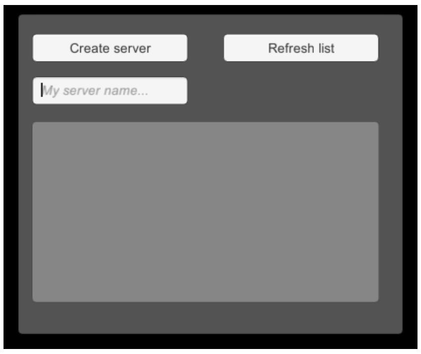
Example

A working example is included in the git repository. Once you've cloned the repository, you can open the project with Unity 4.6 Pro or Unity 5 (any version) and load the MasterServer.unity scene located in: *FIcontent.Gaming.Enabler.GameSynchronization/Assets/Examples/Scenes*.

To use this example with multiple players, you should build the project scene as an application and connect multiple instances together, or in alternative you can deploy it on your devices. The example can also work with a single player in the Unity editor.

Usage

The example takes advantage of the Unity MasterServer to publish the server player address and allow the other players to list the servers and connect to them.



To start a server, just hit the **Create Server** button. This will run a server and publish it on the MasterServer. Once the server is started, wait for the players to connect and then hit the **Start** button to start the Lockstep simulation.

To connect to a server, you can hit the **Refresh list** button and then click on one of the servers that will appear.

Once the simulation is started, every player can move his unit with the arrow keys.

Tutorial

This tutorial is a step-by-step guide that covers the aspects of usage of the Game Synchronization SE by implementing an example project.

Setup the scene

Create a MonoBehaviour script that extends the **FIcontent.Gaming.Enabler.GameSynchronization.AbstractLockstepPeer** class. Create an empty GameObject and attach the following components:

- The script you've just created
- Unity NetworkView component
- FIcontent.Gaming.Enabler.GameSynchronization.LockstepSettings script

Creating IActions

```
using UnityEngine;
using System.Collections;
using FIcontent.Gaming.Enabler.GameSynchronization.Packets.Actions;
using System.Runtime.Serialization;

[System.Serializable]
public class PositionAction : AbstractAction
{
    public const int ActionType = 3;

    private float x;
    private float y;
    private float z;

    public Vector3 position
    {
        get { return new Vector3(x, y, z); }
        set
        {
            x = value.x;
            y = value.y;
            z = value.z;
        }
    }

    public PositionAction(Vector3 point, string guid)
        : base(ActionType)
    {
        this.position = point;
        base.GUID = guid;
    }

    // Deserialization constructor
    public PositionAction(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
        this.x = info.GetSingle("x");
        this.y = info.GetSingle("y");
        this.z = info.GetSingle("z");
    }

    // Serialization method
    public override void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        base.GetObjectData(info, context);

        info.AddValue("x", x);
        info.AddValue("y", y);
        info.AddValue("z", z);
    }
}
```

This code shows how an AbstractAction can be extended to contain the type of data you wish to send over the network. The constructor **PositionAction(SerializationInfo info, StreamingContext context)** is used to de-serialize the data and **GetObjectData** to serialize them. The current implementation uses .net serialization capabilities but you can provide your own serialization by implementing the **IPacketSerializer** interface and provide another way to serialize packets and their actions.

Finally, the **ActionType** member defines the unique identifier of the action and is passed to the **AbstractAction** base class by the constructor. It will be used in the **ExecuteAction** method to distinguish the type of action.

Extending the AbstractLockstepPeer

The following methods need to be overridden:

- ExecuteAction

- OnSimulationStarted

```
protected override void ExecuteAction(FIcontent.Gaming.Enabler.GameSynchronization.Packets.Actions.IAction a)
   {
       switch (a.Action)
       {
           case CreateObjectAction.ActionType:
               var go = Instantiate(playerObjectPrefab) as GameObject;
               go.GetComponent<PlayerObject>().owner = a.GUID;
               playerObjects.Add(a.GUID, go.GetComponent<PlayerObject>());
               break;

           case PositionAction.ActionType:
               var posAction = a as PositionAction;
               playerObjects [a.GUID].Move(posAction.position);
               break;
       }
   }
```

This method casts the **IAction** received to their specific type and takes the action necessary to execute the simulation. In the code above it can receive either a CreateObjectAction that creates an GameObject in the player scene and a PositionAction that moves an object. The actions used in RTS games usually include player inputs (eg: selected units, move units) and game events (unit x attacked unit y).

```
protected override void OnSimulationStarted()
   {
       playerObjects = new Dictionary<string, PlayerObject>();

       AddAction(new CreateObjectAction(this.GUID));
   }
```

This method can be used for initialization. In the example above we create an object for every player in the game.

The **AbstractLockstepPeer.AddAction** method is used whenever the game needs to queue the player actions that need to be sent to the other players.

Managing the connection

There are multiple ways to manage the connections. In the example that is provided in the git repository we used Unity MasterServer [http://docs.unity3d.com/Manual/net-MasterServer.html] class.

You can create a new script that will manage the game creation / connection phase. Once the server address has been defined and given to all the players, the methods exposed by the Game Synchronization SE for establishing a connection are the following:

- **AbstractLockstepPeer.CreateServer()** to start a server on the player that created the game room
- **AbstractLockstepPeer.ConnectToServer(HostData hostData)** to connect to a server. HostData contains IP and port of the target server.