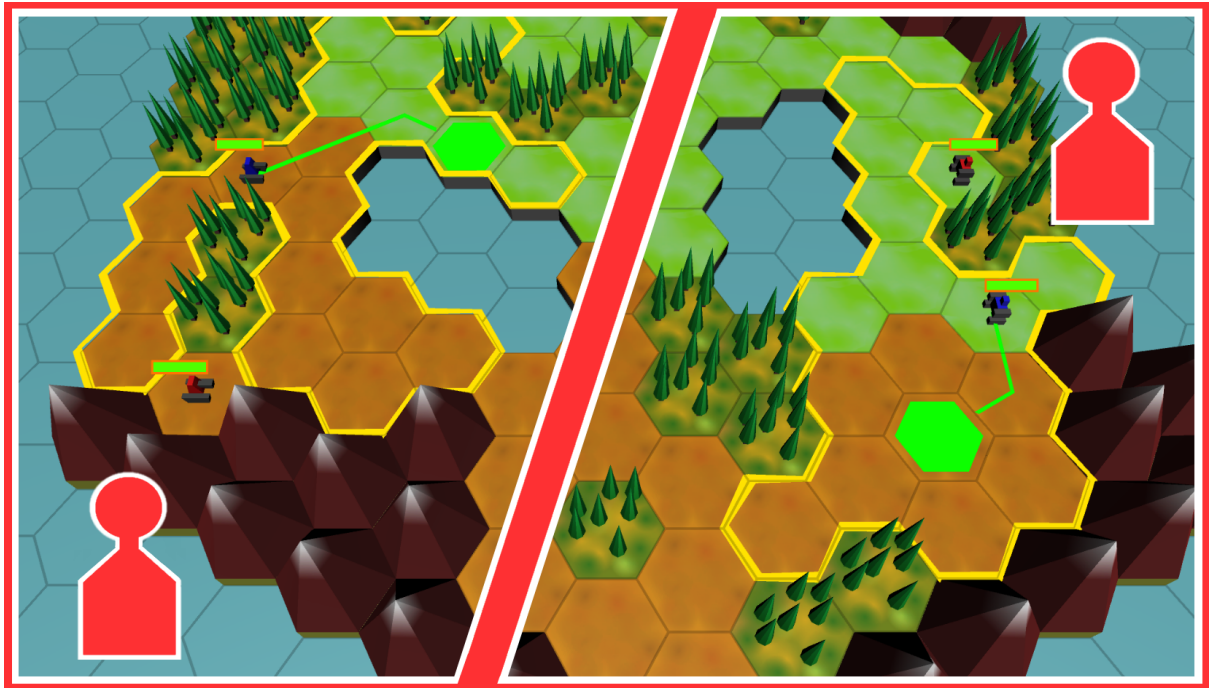


SuperTiles Multiplayer

Documentation



[Join our Discord](#)

Contents

Overview	3
Setup	4
How to run the asset	5
How does it work inside?	8
Gathering in the Room	9
Loading game session inside the Room	11
Playing in the Room	12
Actions	12
Complete turn	12
NetworkTurnPlayer	13
SDK	14
Photon	14
Lobby	16



Overview

This asset extends [SuperTiles - turn based engine](#) asset, which adds Multiplayer support. Information about the creation of level, map and units could be found at this [link](#).

This document will be about the new feature - Multiplayer. Right now, only one network SDK is supported ([PUN](#)). Several new SDK will be added soon.

Current multiplayer implementation uses the idea of **deterministic lockstep**. If every player shares their input with every other player, the simulation can be run on everyone's machines with identical input to produce identical output. It means that messages between machines will be tiny, but care must be taken not to violate determinism. Everything which is important for logic should be synchronized: player list, unit list, random and others. But don't worry, this only applies to new features. All key points in the current asset are already synchronized and just a few steps separate you from your own multiplayer turn-based game!



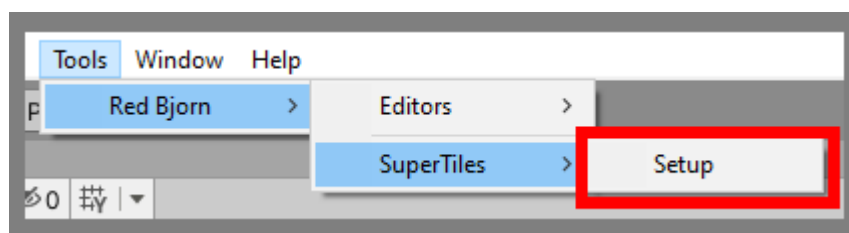
[Join our Discord](#)

Setup

After the importing process of *SuperTiles Multiplayer* asset will be finished, the Setup window should appear.



If the window does not open automatically, you could open it manually



Inside this window, you will see several steps to be done

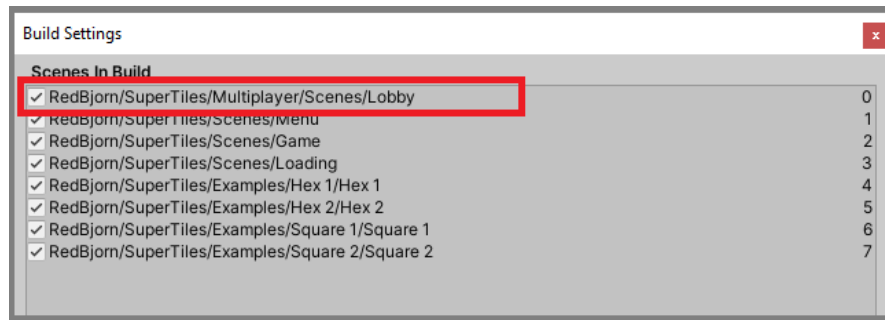
- 1) Import Text Mesh Pro essentials. The asset uses Text Mesh Pro components for UI elements.
- 2) Add asset scenes to Build Settings. This option provides not only valid build settings but also allows you to move between scenes during Playmode.
- 3) Photon SDK. This is an asset which provides the ability of low-level network connections. The corresponding button will lead you to the AssetStore page where you could download it.
- 4) Update items. It is needed only when you upgrade the original *SuperTiles* asset from an obsolete version to Multiplayer version. It fixes several references inside the ItemData assets.
- 5) Create Battle finish conditions. It is needed when you have created levels with *SuperTiles* original asset version. It creates a specific ScriptableObject for every single LevelData.



[Join our Discord](#)

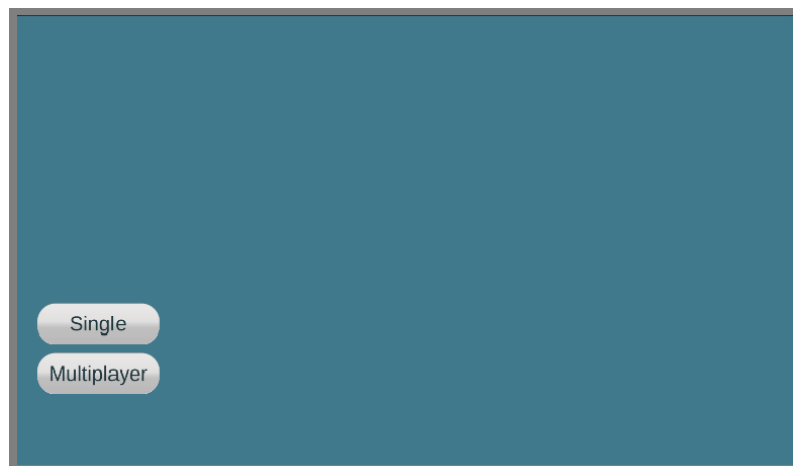
How to run the asset

First, load the Lobby scene. It is the initial scene for the asset.

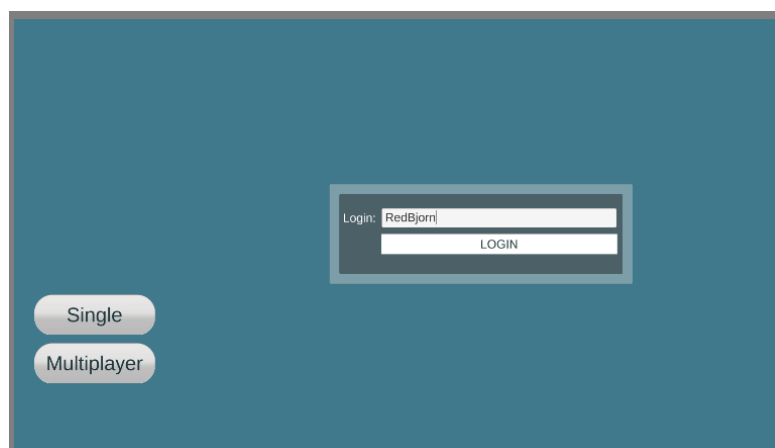


From this scene, you could select

- Singleplayer mode - bring you to the level selection like in Supertiles asset
- Multiplayer mode - bring you to the Login screen

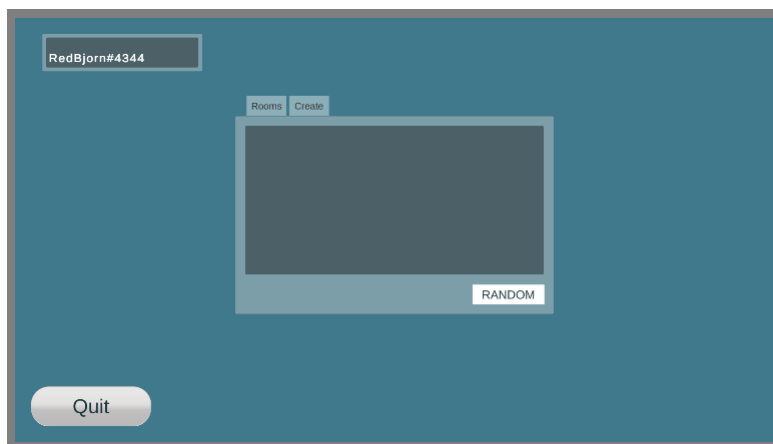


Login could be non-unique. It is more like a nickname. After passing the login screen, you will be moved to the Lobby.



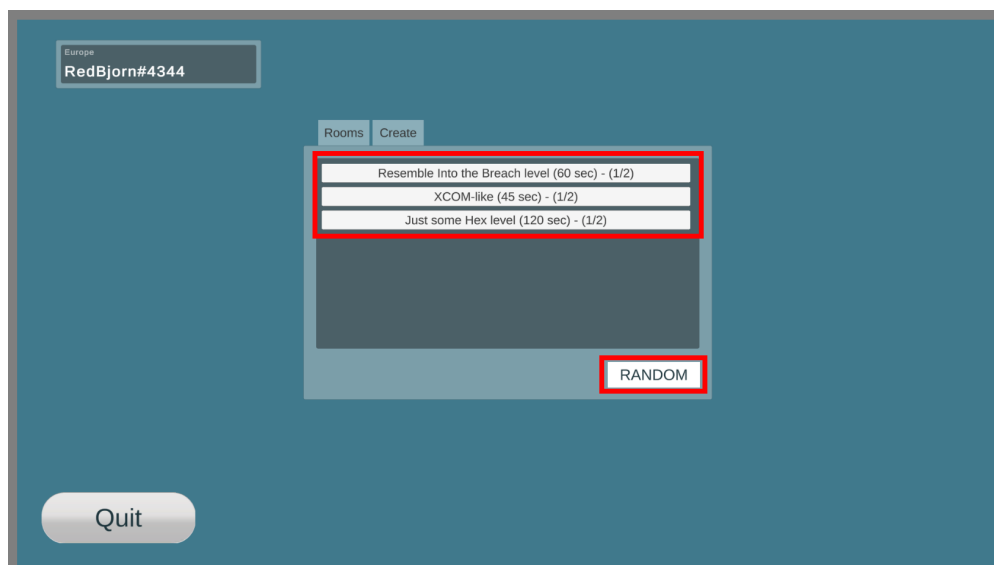
[Join our Discord](#)

Here you can see the Quit button which fully disconnects the SDK from the network and opens the previous menu. At the top left corner, there is your profile panel with a nickname. Four random digits will be added to the nickname to make a visual difference between equal nicknames.



And there are two options:

1. You can create your own session with a specified map at the Create tab
2. You can join already created game at the Rooms tab which is selected by default
 - a. Button Random will take you to a random game
 - b. Or you could join a specific room by clicking the corresponding button

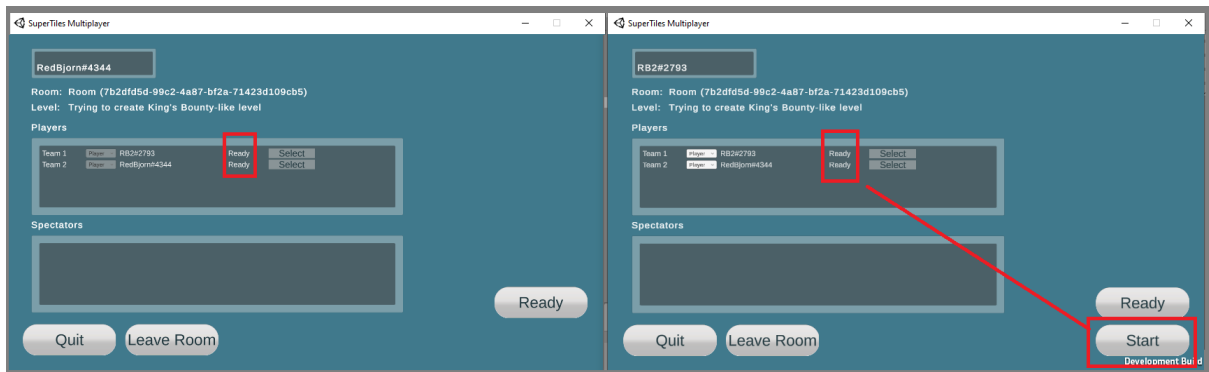


When you join or create a game, you will enter a so-called Room. There is a special role inside the room - MasterClient. This role should not be confused with a host role (a hub for all network messages). Inside *SuperTiles Multiplayer*, there is nothing like a host. We need a MasterClient just to resolve conflicts: change slot type (Player, AI or None), start the Game. If the current MasterClient leaves the room master role will be transferred to another player. If all players leave the room, it will be destroyed, so you can't reconnect to it.

When all players are ready (by changing the current state with the help of the Ready button), the master client could start the game.



[Join our Discord](#)



And.. Play the Game!



[Join our Discord](#)

How does it work inside?

All network logic is encapsulated at the new controller - [NetworkController](#). This controller provides all network methods and callbacks with no SDK-related code.

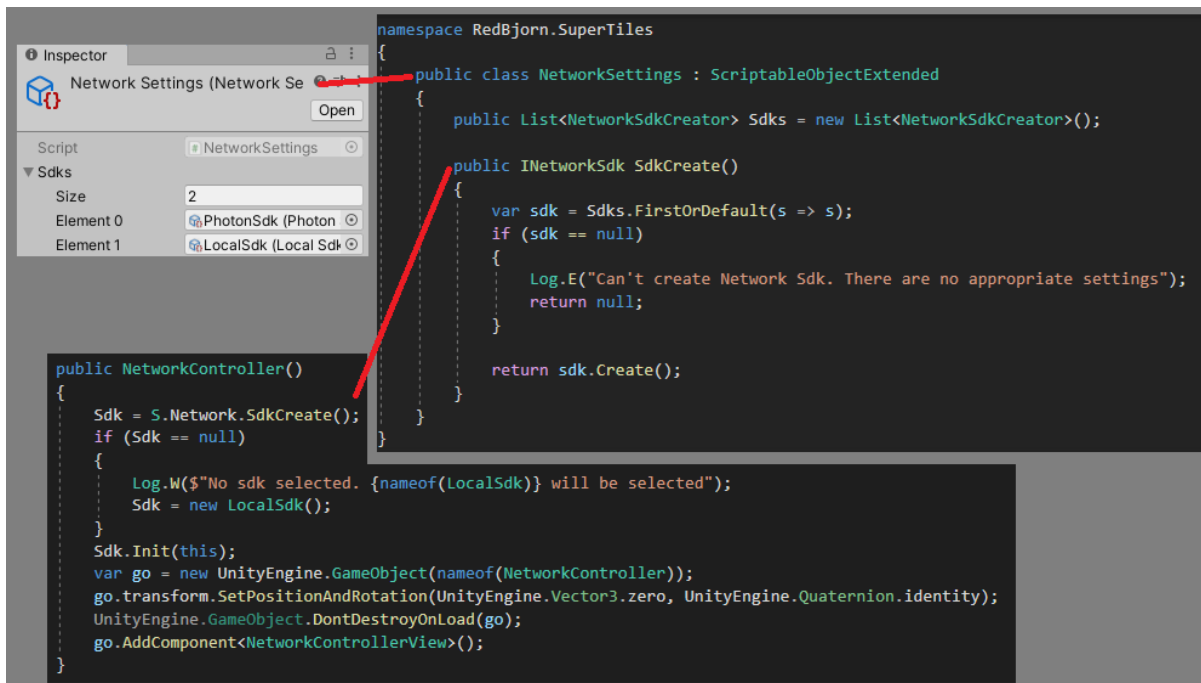
```
namespace RedBjorn.SuperTiles.Multiplayer
{
    public class NetworkController
    {
        public INetworkPlayer Player;
        public RoomData CurrentRoom;
        public List<RoomLobbyData> LobbyRooms = new List<RoomLobbyData>();
        public List<IConnectionCallbacks> ConnectionTargets = new List<IConnectionCallbacks>();
        public List<IRoomCallbacks> RoomTargets = new List<IRoomCallbacks>();
        public List<IIInRoomCallbacks> InRoomTargets = new List<IIInRoomCallbacks>();
        public List<IOnTurnFinishCallbacks> TurnFinishTargets = new List<IOnTurnFinishCallbacks>();
        public List<IOnBattleActionCallbacks> BattleActionTargets = new List<IOnBattleActionCallbacks>();
        public List<ILobbyRoomCallbacks> LobbyRoomTargets = new List<ILobbyRoomCallbacks>();

        INetworkSdk Sdk;

        public const int InvalidId = -1;

        public NetworkController()...
```

[NetworkController](#) creates an instance of the [INetworkSdk](#) interface to communicate with the selected SDK. From the screenshot below, you can see that the SDK will be selected as the first one in the list of SDKs inside the [NetworkSettings](#) asset. [LocalSdk](#) is not another one SDK implementation, it is just a fallback to a singleplayer mode when something goes wrong.



The screenshot displays the Unity Inspector on the left and C# code on the right. The Inspector shows the 'Network Settings (Network Se)' asset with a 'Script' dropdown set to 'NetworkSettings'. Under the 'Sdks' list, 'Element 0' is 'PhotonSdk (Photon)' and 'Element 1' is 'LocalSdk (Local Sdk)'. The C# code on the right shows the `NetworkSettings` class with a `Sdks` list and a `SdkCreate()` method. A red arrow points from the `LocalSdk` in the Inspector to the `LocalSdk` in the `SdkCreate()` method. Below the code, a snippet shows the `NetworkController` constructor logic: it calls `Sdk = S.Network.SdkCreate();`, checks if `Sdk` is null, and if so, logs a warning and sets `Sdk = new LocalSdk();`.

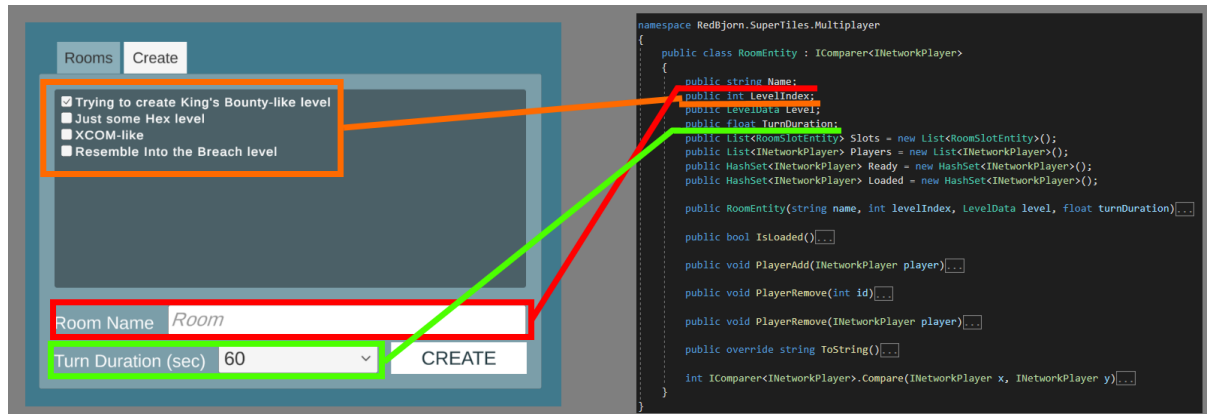
The multiplayer mode can be divided into three stages: [Gathering in the Room](#), [Loading game session inside the Room](#) and [Playing in the Room](#)



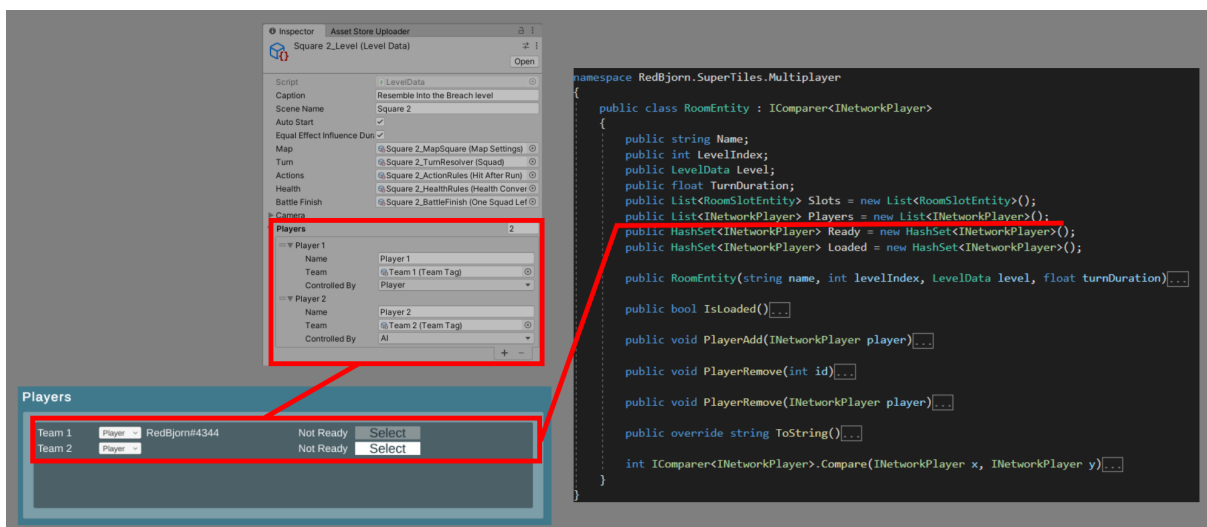
[Join our Discord](#)

Gathering in the Room

At the very beginning, someone creates Room. To do this, it is needed to specify the room name and select the level. Also, you could change turn duration, optionally. This information is stored in a `RoomEntity` instance.

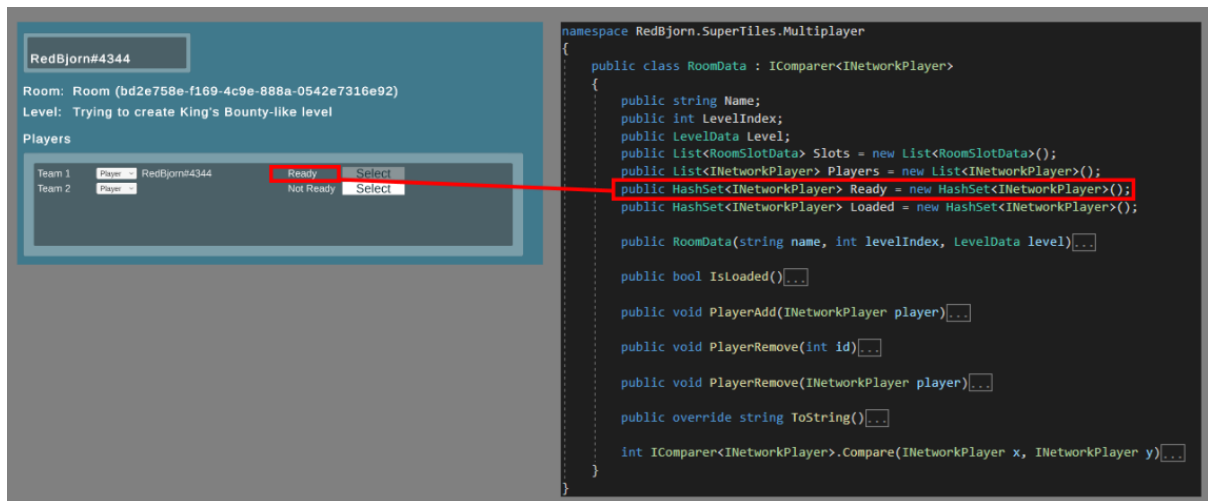


After clicking the Create button, slots for the players will be created from the already configured Players field inside the LevelData asset. AI slots will be replaced with Player slots, but could be easily reverted to the default values.

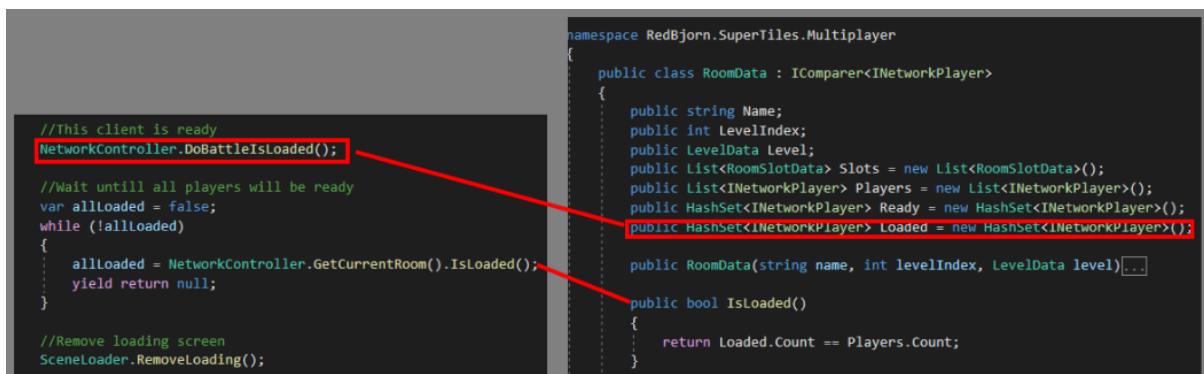


[Join our Discord](#)

Players in the Room could be in two states: **Ready** or **Not ready**. Players which are in the **Ready** state are also stored inside an instance of the **RoomEntity** class.



When all players are ready, then a MasterClient sends a message to everyone that a game could begin the loading process. Level loading should be synced between all game instances. This logic is reflected in the field **Loaded** of **RoomEntity**. When a player loads the level on his side, he sends a message "**I have loaded the level**" (**NetworkController.DoBattleIsLoaded**) and waits when the count of loaded players will be equal to the count of all players in the room.



Gathering stage is completed!



[Join our Discord](#)

Loading game session inside the Room

This stage begins when a MasterClient hits the button Start. This client sends every other player a message to start the game session.



As soon as this message is received, the player loads the selected level from the [NetworkController.LoadLevel](#) method.



The key point here is the Creator field inside the GameEntity instance, which contains a [Multiplayer](#) instance reference. It encapsulates how the session would be created:

- what additional objects need to be created in comparison with Singleplayer regime ([NetworkGameSessionView](#))
- the way [PlayerEntity](#) instances should be created for every game slot inside the current Room
- the moment when the Loading panel should be removed to achieve the same environment for all players



[Join our Discord](#)

Playing in the Room

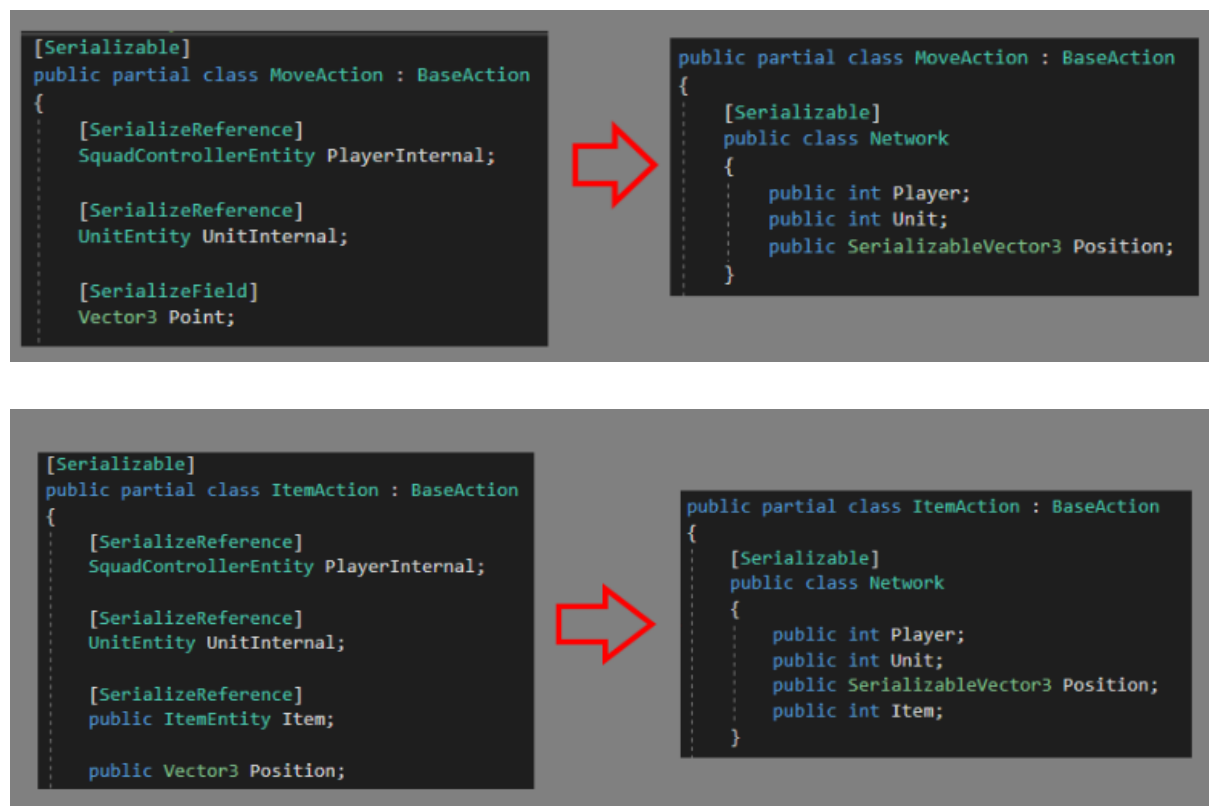
When players have already loaded the level they could create [Messages](#) and send them into [NetworkTurnPlayer](#).

There are only two different message types while playing in the Room

1. Player make an battle action
2. Player completed its turn

Actions

From the [SuperTiles](#) asset, we know actions could be of two types: [MoveAction](#) and [ItemAction](#). To send these actions over the network, we transform standard actions into something appropriate.



By something appropriate, we mean network classes, which could be easily serialized to an array of bytes. Action is sent via [NetworkController.SendBattleAction](#) method.

```
public static void SendBattleAction(byte[] actionSerialized)
```

And such an array can be easily deserialized when it will be received by any other player.

Complete turn

Complete turn message is only a message that contains the turn number to be completed. It is sent via [NetworkController.SendTurnFinishLaunch](#)

```
public static void SendTurnFinishLaunch(int turn)
```



[Join our Discord](#)

NetworkTurnPlayer

It is a specific player for playing actions over the network. When this player receives new action, it decides whether it should be played locally or sent to every other player. Only actions from LocalPlayers would be played locally. This list includes only AI players.

```
public class NetworkTurnPlayer : ITurnPlayer, IOnBattleActionCallbacks, ITurnCallbacks
{
    public class Data
    {
        public BaseAction Action;
        public Action OnCompleted;
    }

    public class SerializedData
    {
        public byte[] Action;
        public Action OnCompleted;
    }

    bool TurnFinishSent;
    bool FinishScheduled;
    double TurnTimeStart;
    BattleEntity Battle;
    Coroutine TurnTimerVerifyCoroutine;
    Coroutine TurnFinishCompletedVerifyCoroutine;
    List<SquadControllerEntity> LocalPlayers = new List<SquadControllerEntity>();
    List<Data> LocalScheduled = new List<Data>();
}
```

Action from a real player will be sent over the network, as was said above, in serialized data format. When this message will be received by NetworkTurnPlayer, it will be deserialized, at first, and then passed to play action logic like it is a simple local action.



[Join our Discord](#)

SDK

There is a core interface of all network SDK - [INetworkSdk](#). It provides a full API for the communication between our [NetworkController](#) and specific SDK.

```
namespace RedBjorn.SuperTiles.Multiplayer
{
    public interface INetworkSdk
    {
        void Init(NetworkController controller);
        void Destroy();
        void Connect();
        void Disconnect();
        bool IsConnected();

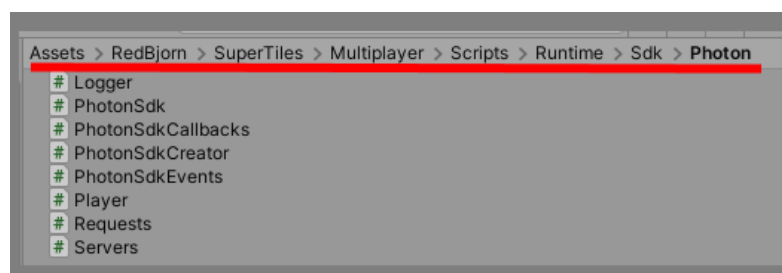
        string DebugInfo();
        double Time { get; }
        string ServerCurrent { get; }
        string ServerDefault { get; set; }
        string[] Servers();
        int Id();
        bool IsMaster(INetworkPlayer player);
        bool IsMaster();
        bool IsInRoom { get; }

        void LobbyJoin();
        void RoomCreate(RoomEntity room);
        void RoomJoin(string id);
        void RoomJoinRandom();
        void RoomLeave();

        void SendRoomSlotChangeType(int slot, int type);
        void SendRoomSlotChangeOwner(int slot, int owner);
        void SendIsReady();
        void SendGameStart();
        void SendBattleIsLoaded();
        void SendBattleStart();
        void SendBattleAction(byte[] actionSerialized);
        void SendTurnFinishLaunch(int turn);
        void SendTurnFinishCompleted(int turn);
        void Error(string message);
    }
}
```

Photon

Photon SDK implementation is based on Photon Unity Networking 2 asset. Core files of this implementation are at the path: RedBjorn/SuperTiles/Multiplayer/Scripts/Runtime/Sdk/Photon.



[Join our Discord](#)

The dominant class is `PhotonSdk`. There is a lot of code inside this class, so it was divided into several

```
public partial class PhotonSdk : INetworkSdk
```

files. First, let's consider the `PhotonSdk` file. This file provides the implementation of `INetworkSdk`. Second, let's move to the `PhotonSdkCallbacks` file. This part of the code handles callback messages received from the `PhotonSdk`. For example, the `NetworkController` sends a connect signal and receives an `OnConnected` message when success occurs.

```
public partial class PhotonSdk : global::Photon.Realtime.IOnEventCallback,  
                                global::Photon.Realtime.IConnectionCallbacks,  
                                global::Photon.Realtime.ILobbyCallbacks,  
                                global::Photon.Realtime.IMatchmakingCallbacks,  
                                global::Photon.Realtime.IInRoomCallbacks
```

Third, `PhotonSdkEvents` is a file for event synonyms. It is recommended to make string events as small as you can, but they can turn into something completely unreadable. That is why we use synonyms. For example, we use `PLAYER_READY_KEY` inside code but send only the `"PR"` string over the network.

```
public const string PLAYER_READY_KEY = "PR";
```



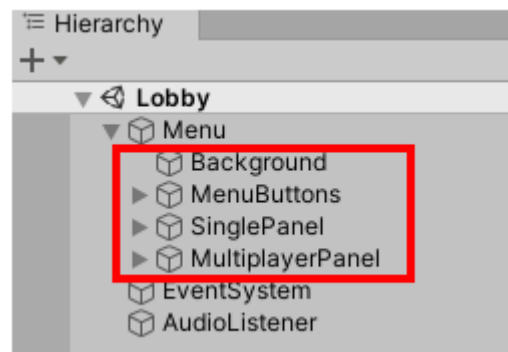
[Join our Discord](#)

Lobby

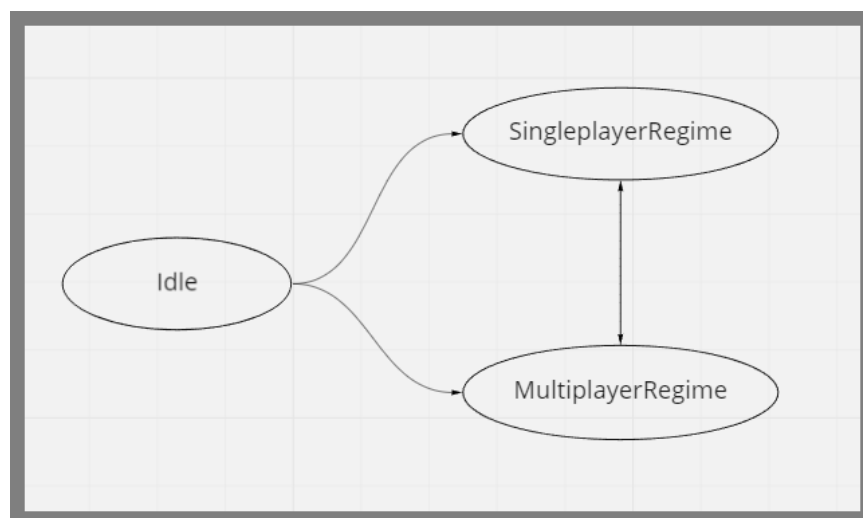
At the Lobby scene, we see only three root gameobjects.

1. Menu - the principal object that will be discussed next
2. Eventsystem - gameobject, which is essential for Unity UI interaction
3. AudioListener - gameobject, which provides the ability to hear music and sounds at the current scene

Menu gameobject is a Canvas which helps to display all the states of the player until the moment the session starts. At the first level we could see regime panels and main menu buttons.

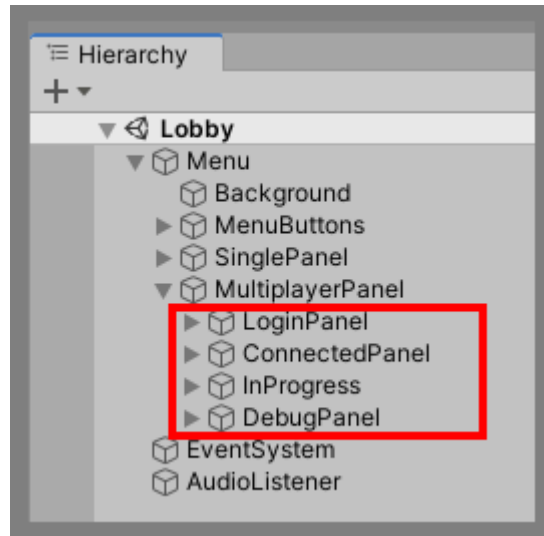


Switching between regimes is represented by a simple State Machine with three states with `MenuUIState` as a base class.



- **Idle** is the initial state of UI. From this state, you could choose Singleplayer or Multiplayer regime.
- **Singleplayer** state provides a level selection panel from where any level could be immediately loaded for the current player.
- **Multiplayer** state opens the panel, which is another independent UI State Machine with `MultiplayerUIState` as a base class.





- ★ **Login** state is the initial state where we should enter a nickname and launch the process of network connection
- ★ **Connecting** is an intermediate state which shows loading animation while the connection process is in progress.
- ★ **InLobby** is a state which we are in after connection is established. Here we have two possibilities: Join room (**RoomJoining** state) or create our own room (**RoomCreation** state). Both these states resemble the **Connecting** state (loading animation during the corresponding process).
- ★ **InRoom** is the final state and the result of joining room and room creation processes. Here we can choose any vacant slot and wait until other players join the room.

