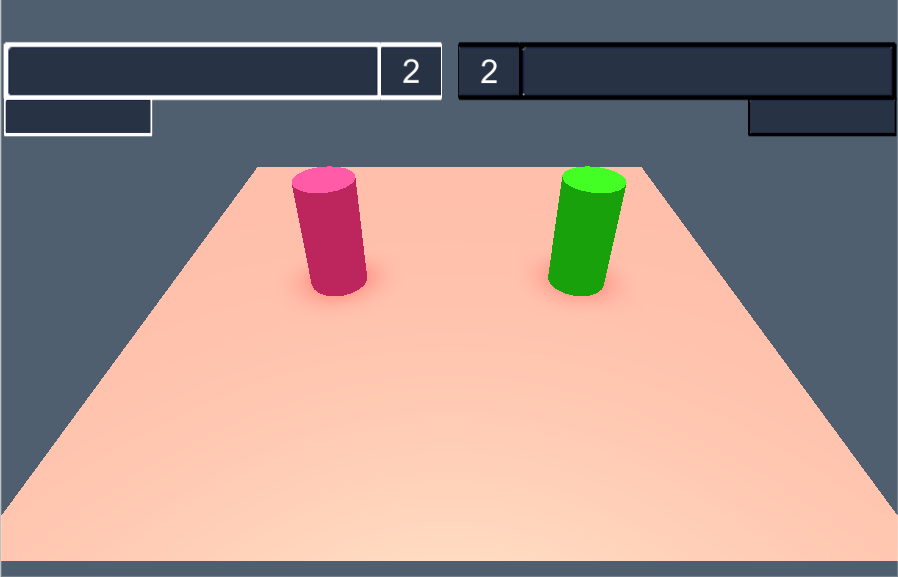
**Unity based realtime game**

In this example we will keep the game extremely simple, but when you understand the basic concepts you will be able to expand on the game logic.

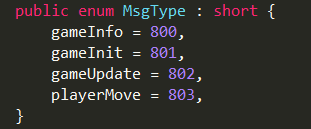
In this game each player will be represented with a cylindrical avatar, that they will be able to move (teleport) around the map.

All of the logic concerning the movements of the avatars is found in the “***BlockBattleLogic***” file, that resides on the “***MainCamera***” object..



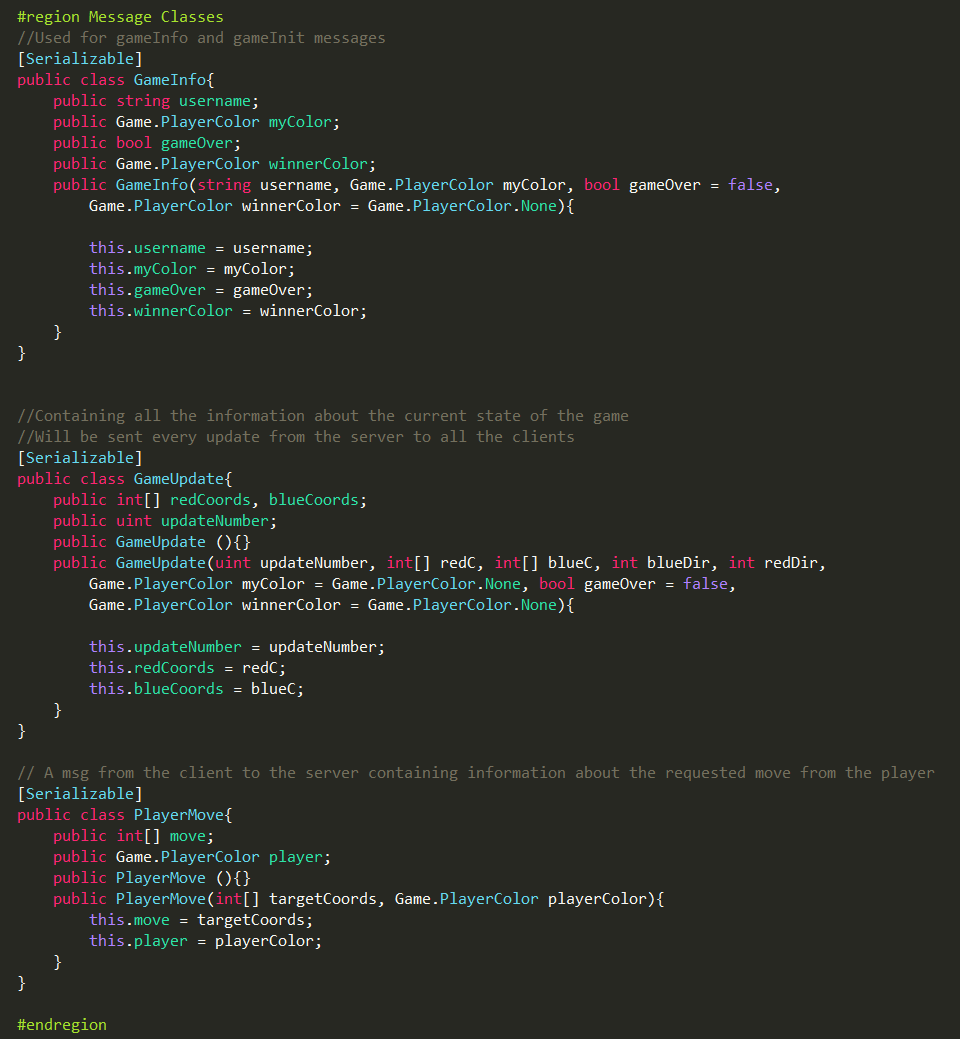
## **Internal Server Protocol** *(Master/GameServer/Client)*

As described in the [protocol](https://docs.google.com/document/d/1pd645gCLWAclsYjqb0omcHK5IR3QdSkxZLz9g5b7VMo/edit#heading=h.s60erl39i6t9) section, we will have an internal protocol that will be used whenever the GameServer and the clients communicate. Here we define what kind of messages that will be sent and their corresponding types(*class*).   
This is done by creating a new script that inherits from the ***Game.CommProtocol*** and will contain this specific games protocol***.***

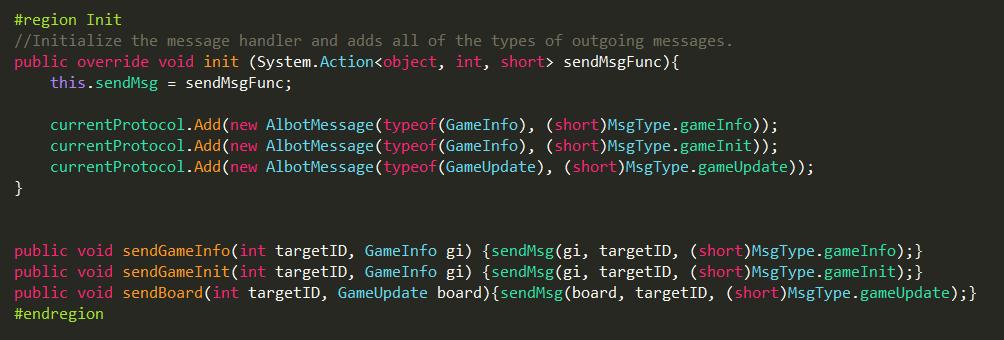
First we must specify what kind of messages that will be sent, and give them an unique ID. This allows us to specify a listener method for every separate ID, and then cast the incoming msg into the specified type. To follow good code conventions, we will bundle this information inside an *Enum*  


* ***gameInfo*Will be sent when the game is over and will also contain information of who the winner is.**
* ***gameInit***Will be sent at the start of the game to every client, containing game information about all the players in the game. Such as their names and color.
* ***gameUpdate***This will be broadcast every update from the server to all of the clients, and contains the current state of the game. Alternatively we could have sent only the latest changes in the game, a method that is preferable if the game state is very large. But for this smal game this is not a problem.
* ***playerMove***A message that will be sent from the clients containing information about the move that the bot from that client wishes to make.

When we have decided upon what kind of messages that will be sent, we must specify the format of these messages and their attributes. This is done by creating a class that will contain the necessary message information. An object will then be created from this class, serialized into a byte msg, and finally sent over the network. For this game we will create three different message types.



As you might have noticed we have specified four different kind of messages, but only created three message types. This is because we will for the sake of code reduction re-use the ***GameInfo*** class for both the ***gameInfo*** and the ***gameInit*** message.

Finally we must init the game protocol and declare these messages to Albot, this is done by the following code:

In the ***init*** function we init the sendMsg function and declares our outgoing server messages to the Albot protocol.   
  
Here we also create an internal interface for the server, where we specify a protocol functions for the outgoing messages. This reduces code repetition later on but more importantly it ensures that we don’t accidently send a message with the wrong label.

## **GameType** *(Master/GameServer/Client)*

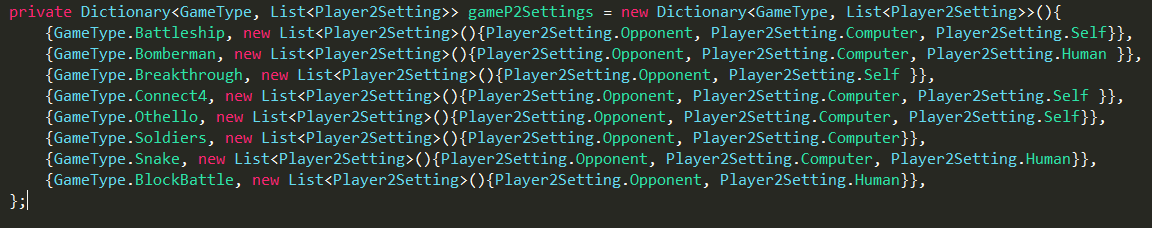
In the file ***GameMaster***there exists a enum ***GameType***, here we need to enter our newly created game. This again serves the purpose of relieving us from typing the actual name out in the future. Diminishing the chances of certain typos.

If the game we have created is a multiplayer game, meaning that it runs on the server then we also have to att a case in the ***GameUtil.strinToGameType***. This is used in the startup of the GameServer. Since all arguments are sent to the GameServer as commandline arguments, we need a way to incorporate the string as a GameType.

**\*\*\*** OBS, **Make sure that the string is in all-caps!!! \*\*\***

## **PreGameLobby***(Master)*

In the file ***PreGameLobby*** there exists a dictionary ***GameP2Settings***. This is where we specify what types of opponents will be available when we create a new game. Since this is a realtime game we will not inclunde Self. Since this is not currently supported for realtime games. We will however add the Human option, so we can easily try out the games for ourselves.



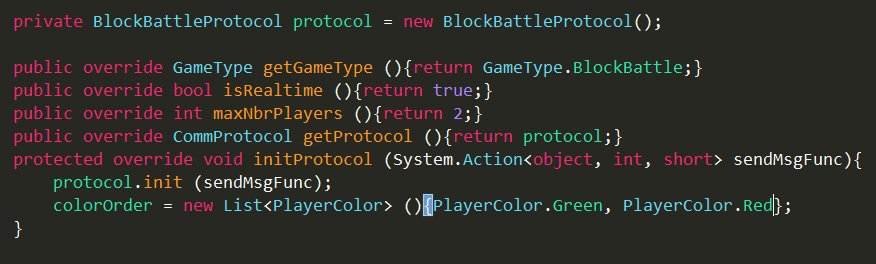
## **GameSettings***(Master)*

In the file ***GameSettings*** we also need to make an entry in the ***InduvidualGameData*** Dictionary. These arguments should be quite self-explanatory.

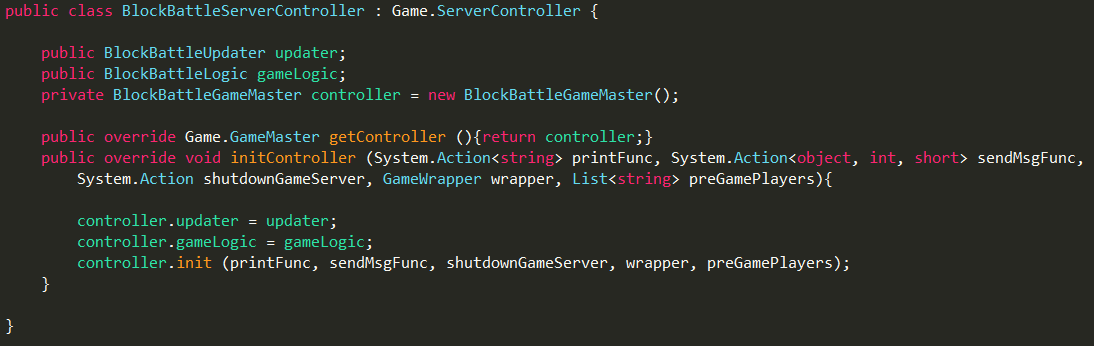
## **GameMaster** *(GameServer)*

In the previous section we added some constants to the ***GameMaster*** class, this is the class that manages a lot of the backend server communication and acts as programming interface for us the game developers. We will therefore create our own ***BlockBattleGameMaster*** and set it to inherit from the original ***GameMaster.***

First there are some setting which we need to set, for our demo we will configure them as follows:



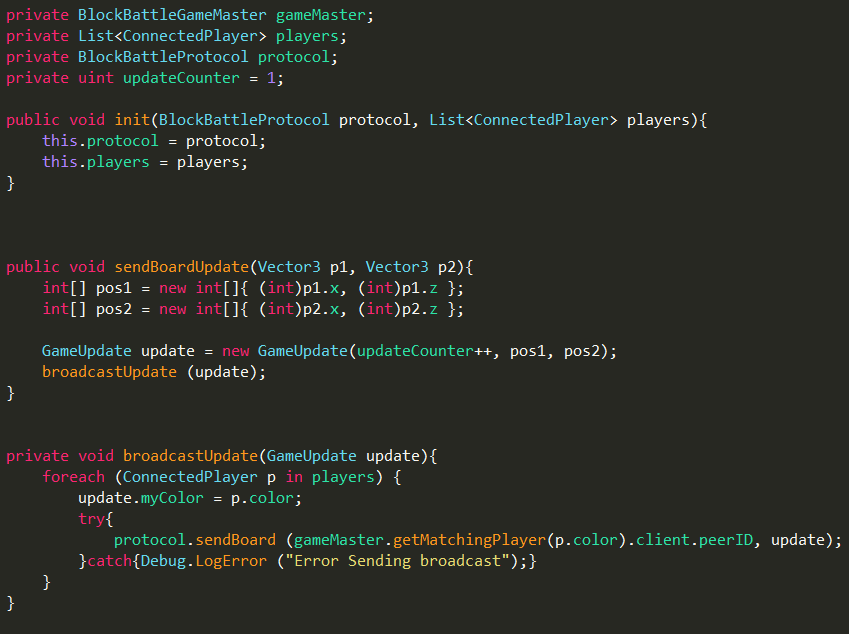
First we create an instance of the protocol that we created earlier. When the GameServer is started it will call the ***initProtocol*** function and pass the ***sendMsgFunc*** that allows our script to communicate with other players.  
We also specify the ***colorOrder***, this is the order that we will distribute colors to the players as they are ordered in the preGame. Currently we give the game creator the color ***green***, and the second player will play as ***red***. Due to convenience players are often referenced using their colors.  
  
The rest of the settings are quite self-explanatory. We will return to our gameMaster later to define how the game will start and what happens when players leave the server. But for now let’s set up the GameStateUpdater.

**ServerController***(GameServer)*  
We now have to create the ServerController for our game. The ServerController is the first object on our GameServer that is initiated, and it will be here that we create our instance of the GameMaster.   
Since ***GameMaster*** is not a Unity-component we will send references to it in the ***initController*** function. For the sake of this tutorial we will lazily set the ***updater*** & ***gameLogic*** references over public variables.  
  
We now need to create the actual Updater we just sent an reference to.

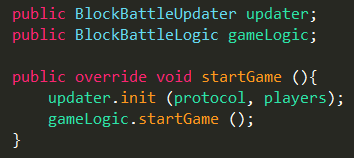
## **GameStateUpdater** *(GameServer)*

The updater is the component that looks at the current board, formats it and then broadcasts it to all players. Most time the only complex here is how you choose to capture the current game state. We will go with a markov approach and simply send the current position of both players.

As I’m writing this tutorial i realize that most of the code is boilerplate-code, and we should find a clever solution to inherit/implement most of the repeated logic. But for now we will let ur updater have the following code:

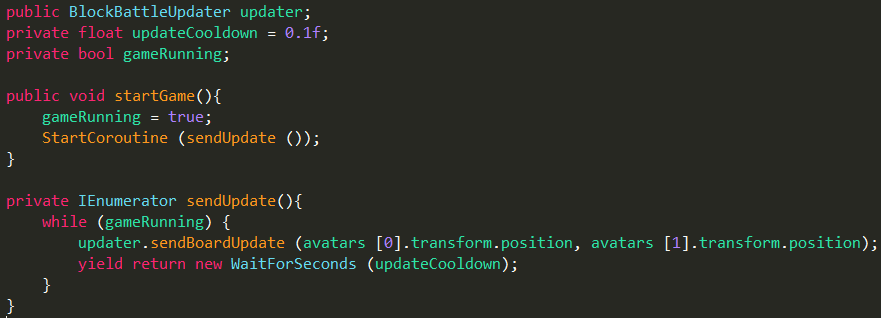
  
  
The init function will be called from the ***GameMaster***that we created earlier and ***sendBoardUpdate*** will be called from ***GameLogic***.

To keep up we make the following additions to our ***BlockBattleGameMaster:***

**

## ***GameLogic - 2*** *(GameServer)*

We will now add the following code to the our ***GameLogic.***

**

Hopefully this is quite self-explanatory, but something that can be noted is the use of the IEnumerator. We could just as easy have used the classical ***Update*** function that we inherit from ***MonoBehaviour***. But since this will be running with a fixed interval it’s convenient to have it run in an coroutine.

It’s now time to leave the GameServer and setup the client side of our game. We will however return later to handle player messages from the clients. And then we will pretty much be done.

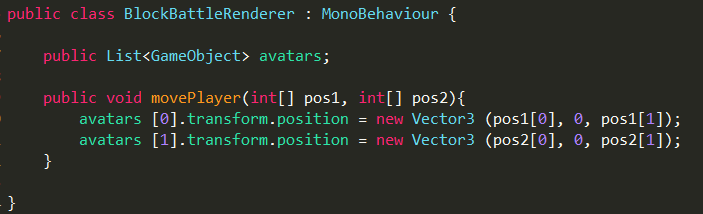
## **Creating the Game Client** *(Client)*

The client side of a game in Albot can be structured up into the following models:

* ***Renderer***This is a umbrella term for all the visualisations that concerns the game logic. In essence the renderer receives a game state from the server and then displays it. This component is highly game dependant and will vary widely from game to game.   
    
  For turnbased games it’s usually quite straightforward, for realtime this might be more tricky. A good example of this is the soldiers game, here we receive approx 10 updates per second. And then we interpolate in between these states. So that we display a smooth game.
* ***ClientController***Much like its server counterpart the ***ClientController*** is the first thing that is initialized at the startup of the game. After the startup phase it will act as a router regarding all communication with the ***GameServer***.
* ***ClientController (****TCP-Handler****)***Embedded inside the ***ClientController***is the ***TCP-Handler***, its sole purpose is to handle the communication with the player bots. When creating games we will however use the ClientController as a interface to the TCP messages.
* ***PlayerEntity***  
  The ***PlayerEntity*** is the object that will contain the ***ClientController***, in fact it will contain every controller for all of the games. The ***PlayerEntity*** object is dynamically loaded into the game, meaning that it **has** to be located in the ***Resources*** folder!!!!  
    
  When a new game is created in the client the ***PlayerEntity*** is always spawned. It will thereafter iterate through all of its controllers and pick and init the one matching the current game type.

## **Renderer** *(Client)*

The ***Renderer*** for our game will be an identical replica of the the server game logic. This is is however rarely the case. Usually we wish to do more complex rendering at the Client. For this game we could easily add interpolation between teleoperation states. But we will skip it for this tutorial.



The ***Renderer*** simply receives the positions of the two players and instantly moves the avatars to the specified position.

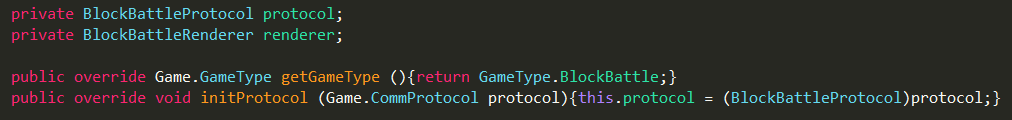
Something to note is that the ***Renderer*** component need to placed on the ***GameController*** objcet. This is so that the ***ClientController*** can find it during startup.

## **ClientController***(Client)*

As mentionend [earlier](#_l8eb59ym68rs), the ***ClientController*** component needs to be placed on the ***PlayerEntity*** object. We will therefore go to the ***PlayerEntity*** object and create a new controller that inherits from ***Game.ClientController.***

### **Settings:**

Right off the bat there is some settings that we can setup:

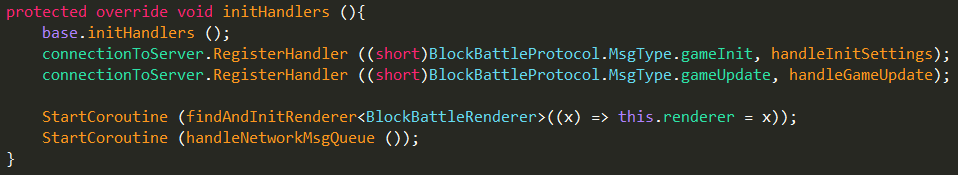


As mentioned the correct ***ClientController*** is selected based upon what game type it has. We therefore need to specify the ***getGameType*** function. When a controller has been selection two of its functions will be executed: ***initProtocol*** & ***initHandlers***.

***InitProtocol*** is quite straightforward, we will receive the current version of the protocol although we will need to specifically cast it to our specific game type.

### **initHandlers:**

The initHandlers function is where we specify how we will handle all of the incoming server messages. So far the initHandlers function have also been used as a generic Start function for the controller. Meaning that if there is something else that needs to initialize we do it here.

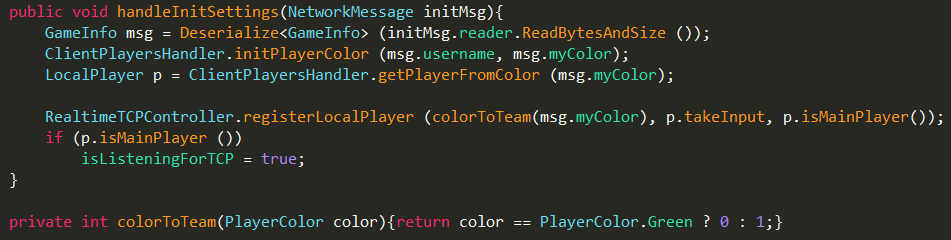


Make sure that you call the ***base.initHandlers()***, it will automatically setup handlers for the ***playerJoined*** event. Then when a ***playerJoined*** message is received, it will automatically display that player name on the ***GameUI.***

After this is done we can specify a custom handler for a message type. This is done by calling the ***connectionToServer.RegisterHandler*** and declaring what type and function. As you can see we have created a a handler for both the ***gameInit*** and ***gameUpdate*** messages, we thus need to create those functions.

Finally we do some other initalistation of the game. First we use the coroutine ***findAndInitRenderer***, that searches the ***GameController*** object in the scene and returns the component matching the given arguments. We also start the coroutine ***handleNetworkMsgQueue***, this is coroutine that will send our outgoing messages to the ***gamServer.***

### **handleInitSettings:**



All handlers receive an ***NetworkMessage*** which we can then cast to the specified message type. Since we have specified the protocol, we can be sure that the message will actually be of the desired type.

As we declared in the [Internal Server Protocol](#_21o3d07lhxbz), the init message will be sent to every player that has joined the game. This means that one client can be sent several init messages, one for every player that is controlled by that client. For example if we choose to play against a training bot, our client will run both of these, and we will thus need to handle two init messages.

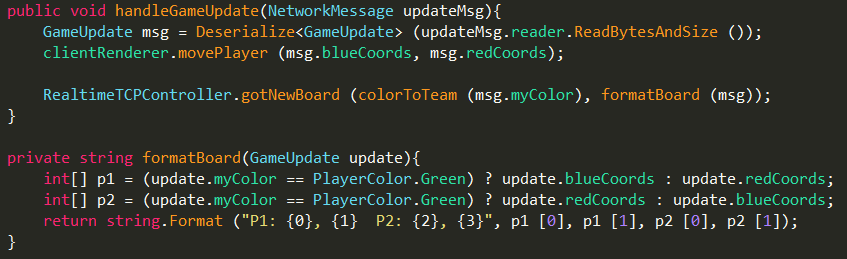
The ***ClientPlayersHandler*** contains stored information from the ***preGame*** of what player corresponds to what color and their role*(Player/computer/human)*. This functionality will probably be moved to the server and sent along the message, but for now we store this information locally.

Based upon the color of the player we decide upon the team. When we receive a game update we will store it, so that when a player makes a move it will instantly receive the latest stored update. We can format updates differently for every team, which comes very in handy for games like **Connect4** or **Othello**.

We thereafter register the new local player at the ***RealtimeTCPController***.r.

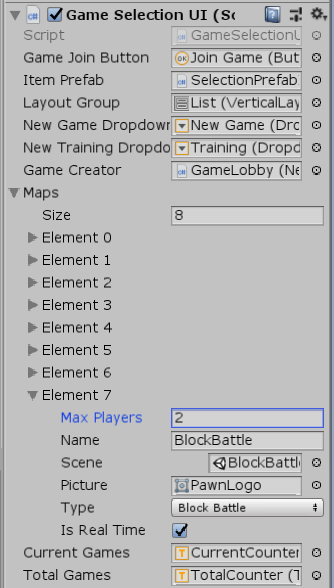
Finally if the player is a TCP player we start listening for TCP messages.

### **handleGameUpdate:**



## **GameSelectionUI***(Client)*

We have now come so far that we are prepared to test-play our game, but to do so we need to be able to start it from the client side. This is done by adding it to the possible games in the gamelobby.

We therefore head over to the Client Scene, and find the ***ClientLobbyUI->GameLobby* object**. On this object there exists a component named ***GameSelectionUI*** and this is where we will add our new game. We simply add one to the **maps** size, and adjust the following settings:

* ***Size***Add one to the current size, to give us a new game entry.
* ***Name***The name of the game.
* **Scene**  
  Should contain the Client game scene.
* ***Type***Select the type for the new game.
* **Is Real Time**Check this, since our game is realtime.

Make sure that you get all of these settings correct!

## **New Game Dropdown***(Client)*

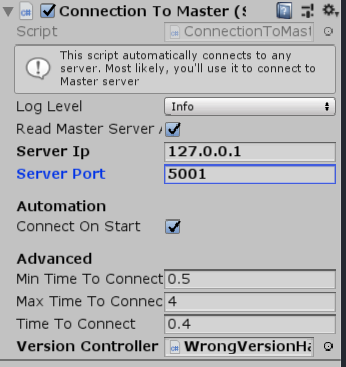
Now that the game exists as an option we need to make a entry in the ***NewGame*** list. Locate the object: ***ClientLobbyUI->GameLobby->GamesWindow->Buttons->NewGame.*** This object contains a dropdown component, which in turn contain list of available options to click. Add a entry here to represent our new game.

**\*\*\*OBS\*\*\***

Make sure that the order of the buttons in the dropdown menu matches the order on the ***GameSelectionUI.*** Atleast for now the selection is index-based, meaning that button 3 selects element 3.

**\*\*\*\*\*\*\*\*\*\*\*\***

When a new item is added you will probably need to adjust the position of the ***NewGame->Template*** object to match with the new entry. This is easiest achieved by letting the Template object remain Inactive, make minor adjustments and try it out by playing the Client scene. At Least in Unity 5, the Template object would behave very irrational if you adjusted it whilst it was active.

**IP/Port Settings***(Client)*

Although it’s possible to set the parameters of how clients/spawnerns will communicate with the ***MasterServer*** using command line arguments. It’s much easier to configure all our three Albot online components to talk to each other in the editor.

In the client scene, find the object ***ConnectionToMaster.***

This component also exists in the MasterServer/GameServer scenes, so make sure that when you do make changes to this component in one of these scenes, you click the **Prefab**->**Apply**. This will synchronize the settings in all your scenes.

* ***ServerIP***This needs to match the IP of whatever device is hosting the ***MasterServer.***
* ***ServerPort***Can be any arbitrary chosen port that is available and open.

When you have have decided upon a port and IP, these changes also need to be added to the ***MasterServerBehaviour*** and ***SpawnerBehaviour****.* Both of these components are found in the ***MasterServer*** scene.

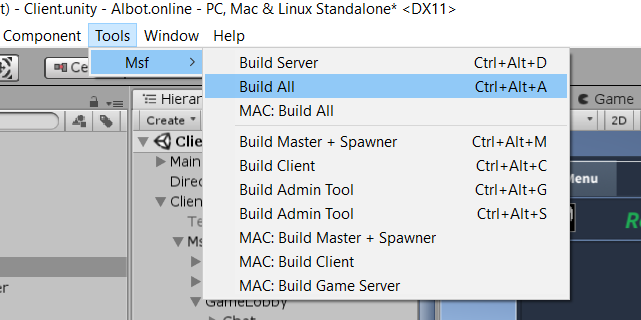
* **MasterServerBehaviour***Simply make sure that the port matches the* **ConnectionToMaster** *port.*
* **SpawnerBehaviour**Make sure that the IP address matches the**ConnectionToMaster** address. You also need to specify the name of your ***GameServer*** executable.

**Build Settings***(Client)*

Finally before we can try our game we need to build our executables. The ***MasterServer***, ***GameServer*** and ***Client,*** although it’s possible to run the client from the Unity editor. To make it easy for our future selves, we will add our changes to the ***msfQuickBuild*** script, and automate the build settings.

* ***MasterServer***We have not done any changes to the main server logic except added our new game in the ***GameType*** enum. Therefore like always the only scene needed for the ***MasterServer*** is the MasterServer scene
* ***GameServer***Since the game that we have created is Unity based, we need to add the ***BlockBattleGameServer*** scene to the GameServer build.
* ***Client***Just like on the ***GameServer*** we need to add the game scene, however in the ***Client*** build we add the client game scene.

Open up the ***msfQuickBuild*** script and add an entry in both the ***buildClient*** an ***buildGameServer*** functions. We now have the ability to use the quick builds, and it will automatically include our new game.



The quickbuilds is located in the menu ***Tools->Msf***, alternatively you can ofcourse use the shortcuts.

If you do decide to have the client side run in the editor you need to add the Client game scene to your build settings as well.

## **Playing the game**

We are finally ready to play our game! Here is a simple python bot that moves our player to a random location, waits 1 second and then repeats.