Multiplayer Battle Wizards

v1.0

Unity Asset Pack

Contents

Project Overview

Multiplayer Battle Wizards is a complete, 2D multiplayer fighting game in Unity. With up to 4 players, you can battle each other using unique spells in many different arenas.

Features

- A complete, multiplayer game using Photon.
- An easy to use map editor.
- Mobile support.
- Fully documented code.
- Customizable spell creation.

Scenes

- **Menu** - the starting scene of the game and where the players can lobby up, change settings, etc, before entering the game. Always launch the game from this scene.

- **Game** - where the action takes place. When players load into this scene, the map is spawned in and many other things are initiated. Do not start the game from this scene, as there will be errors and nothing will happen.

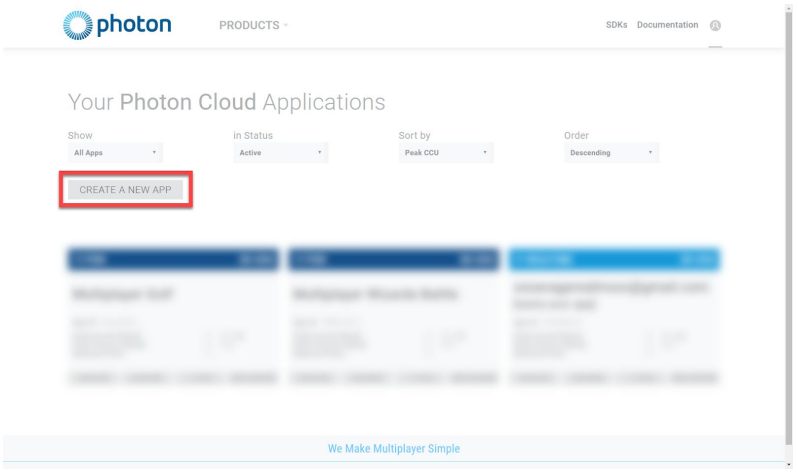
- **MapEditor** - a tool you can use to create and edit maps for the game. Don't include this in the build settings as it won't work outside of the Unity editor. Learn more.

Photon Setup

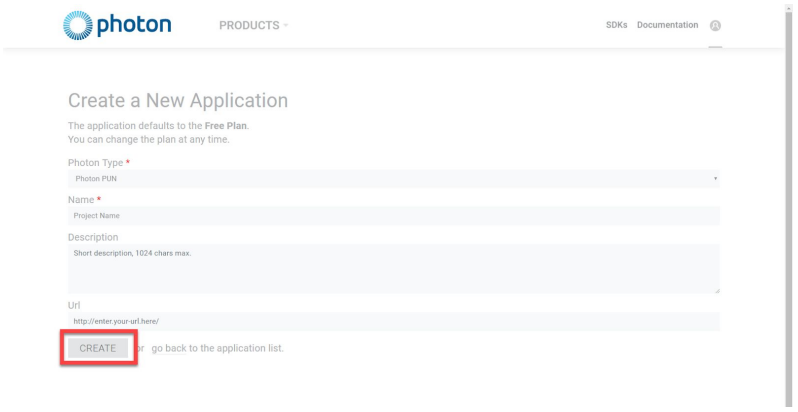
Photon is the networking framework used in this project. In order to use it though, you need to sign up and create an app. This is because along with being free, Photon does feature paid services if you wish.

1. Go to https://www.photonengine.com/pun and sign in or create an account.

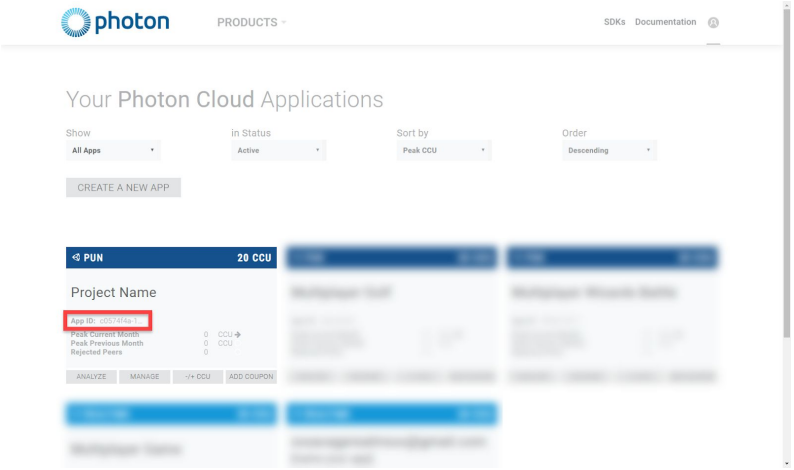
2. Go to https://dashboard.photonengine.com/en-US/publiccloud and create a new app.



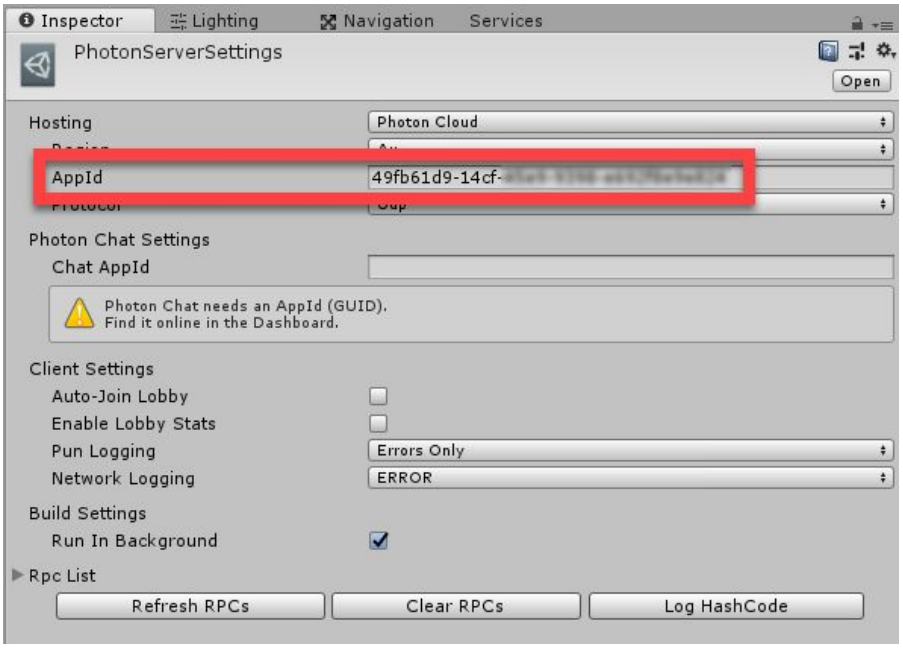
3. Select **Photon PUN** and enter in a name for your app (project). A description and url are optional.



4. Back on the applications page, you'll see your new app in the list. What we want from here, is the **App ID**. Copy that, then let's go into the Unity project.



5. In the Unity project, we want to go to the **Photon Server Settings** asset (*Window > Photon Unity Networking > Highlight Server Settings*). Paste your **App ID** into the field.



In the **Photon Server Settings**, you can also change the region of where the servers are located. Select the location you live in (this can be changed in-script).

If you're having issue with Photon or the files are corrupt in some way, you can import Photon into your project from the Asset Store.

Network Structure

How it Works

Unity networking and Photon allows the client and server to communicate to each other through **RPC**'s (remote procedure calls). These are basically functions that you can call and choose who else in the game calls it. For example, if the player runs into a powerup and you want a colourful effect to play, that player would send all of the clients the function to perform that effect.

Where multiplayer games become tricky though, is when you're deciding what information the client will send to all others, and what info they'll keep to themselves.

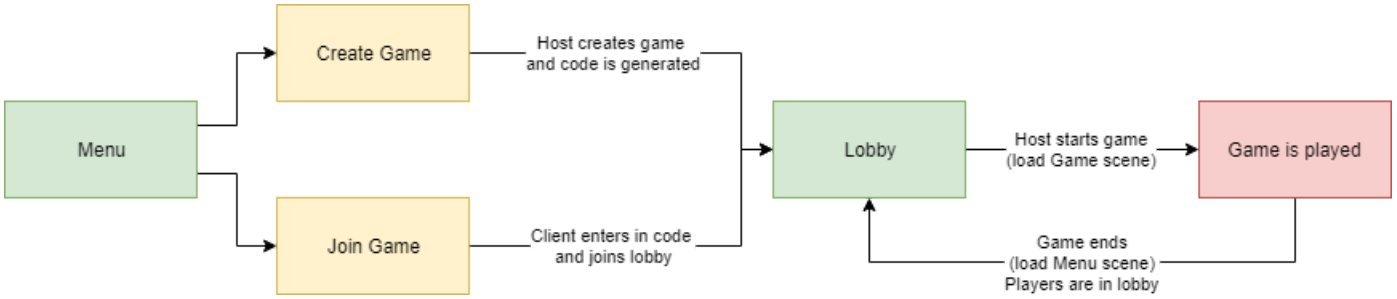
As an example, if you got a powerup that gave you more damage - would you tell all the clients that your damage is now increased? They don't need to know. The only time they'll know, is when they get hit by you - and that would be done with you sending over a call to them containing the damage to be dealt.

This is a problem many developers face. Having a balance between lowering the amount of calls each second, and having a game that is not easily tricked or hacked.

Creating / Joining a Game

The game uses Photon as the networking framework. When the game starts it loads up the Menu scene and attempts to connect to the Photon Master Server. The menu buttons are inactive before and during this process and are only usable once the connection to the server is made.

Each game which can hold 4 players are known as *rooms* in Photon. These are seperate games or matches that you can create. Below is a diagram showing how the network runs in the game:



In the menu, the player can choose to **create a game** or **join a game**.

- **Creating a game** will do the following:
    - Create a new *room*
    - The name of that room will be a randomly generated 5 character code (e.g. DJ3N1, 27LN8, etc...)
    - The *lobby* screen will appear and show the players currently in the game (only the host when they make it)
    - **The player who created the game is known as the *Host (aka Master Client)***

- **Joining a game** will do the following:
    - Open the *connect* screen
    - The player can then enter in a room code
    - The server will try to find a game with the name of that room code
        - If it does, the player joins the lobby
        - Otherwise, an error will appear

Once a *room* is created and the players are in the *lobby*, the host (and only the host) can choose to start the game. Thus, prompting to change the scene to the Game scene.

If the host ever disconnects from the room, a random player in that room will become the new host.

Loading into the Game

Once the host starts the game in the *lobby*, the Game scene will load. When a player loads into the scene, they will be flagged as *inGame*.

The host will be checking to see if everyone is *inGame*. If they are, then the respective players will spawn and be able to take control. If not, then the host will continue to check.

Once all the players are spawned in and able to move around, the host will trigger an on-screen countdown on all the client's screens.

After that, combat is enabled and the respective gamemode begins.

Gameplay

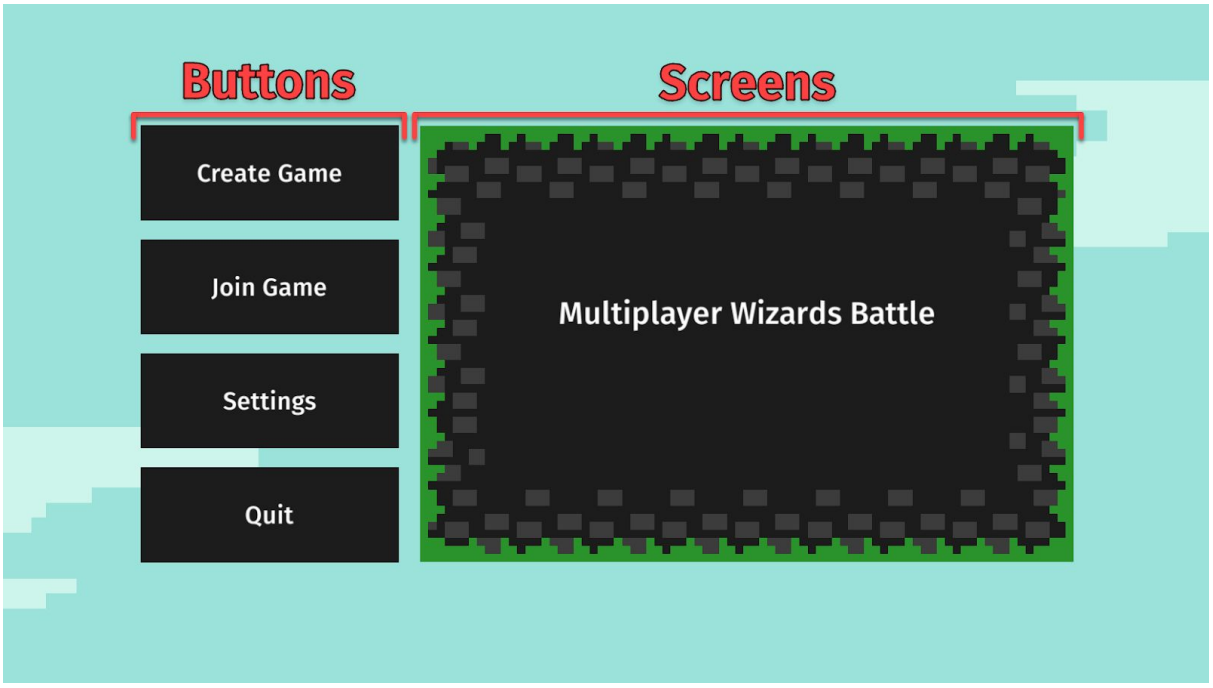
There are two ways mechanics and events are passed through the server. These are **client side** and **server side**.

Client side is where things are triggered when they happen on the client's computer. For example, you may of heard the term *client side hit detection*. If a player shoots a spell at another player and hits them - this is client side hit detection. Where as, **server side** hit detection occurs in a game where the projectile is only detected if it so happens on the server.

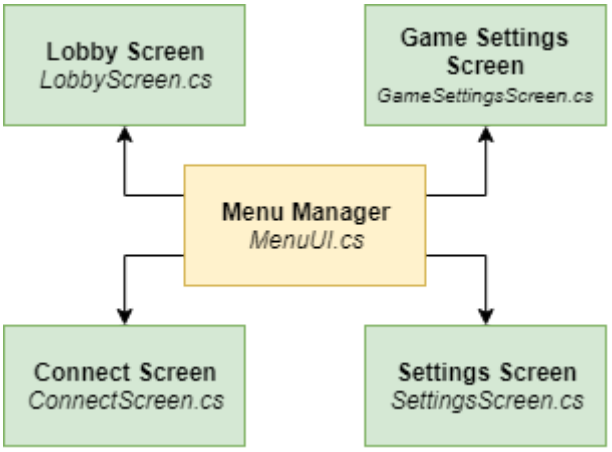
Client side allows for more leniency on your side (if you hit them, you hit them). Where as, server side allows more leniency for all the other players. In this project, we're using client side hit detection, yet other areas like picking up pickups are done through the server - so that no two players can pick it up at the same time on their screens.

Menu

The menu is where the player will find themselves after launching the game. Located in the *Menu.unity* scene, here's what it looks like.



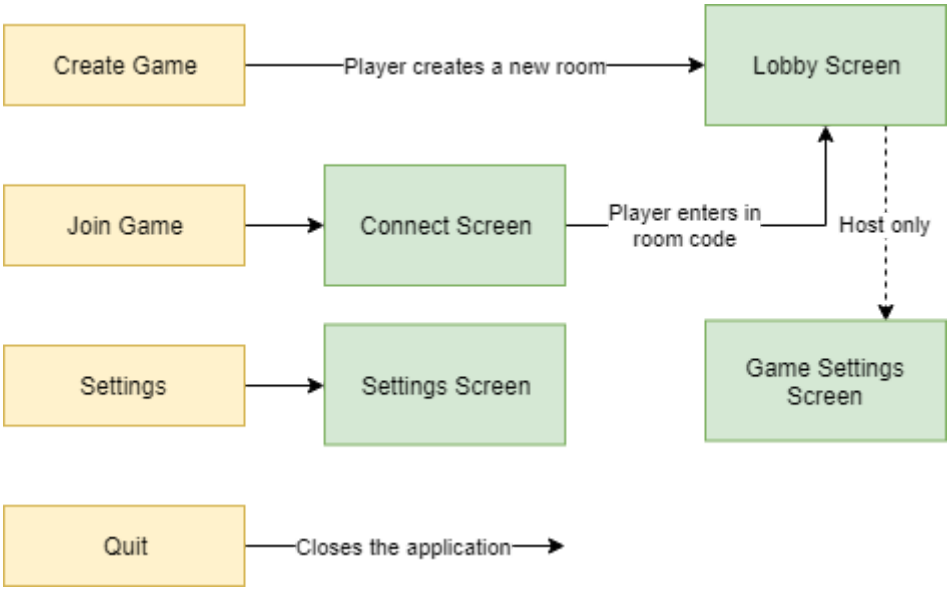
There are two main areas on the menu. The **Buttons** are a set of 4 buttons that the player can click on at any time to navigate / do their specific function. The **Screens** are a collections of different panels / windows / pages - each with their own purpose.



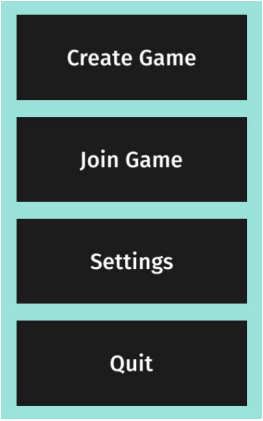
The menu is managed by the **MenuUI.cs** script.

This connects to all the other scripts managing their separate screens. If you want to change screens or toggle buttons, you need to go through this hub script.

Here’s an overview of the menu and how the buttons and screens connect.



Buttons



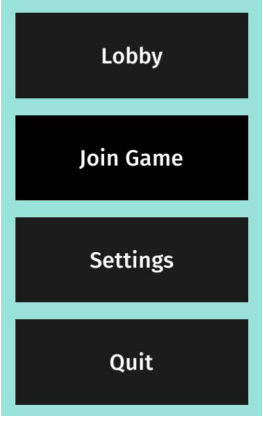
Create Game - creates a new room and sends the player to the lobby screen.

Join Game - opens the connect screen where the player can input a room code.

Settings - opens the settings screen.

Quit - quits the application.

When the player is inside of a lobby, there are a few changes to the buttons.



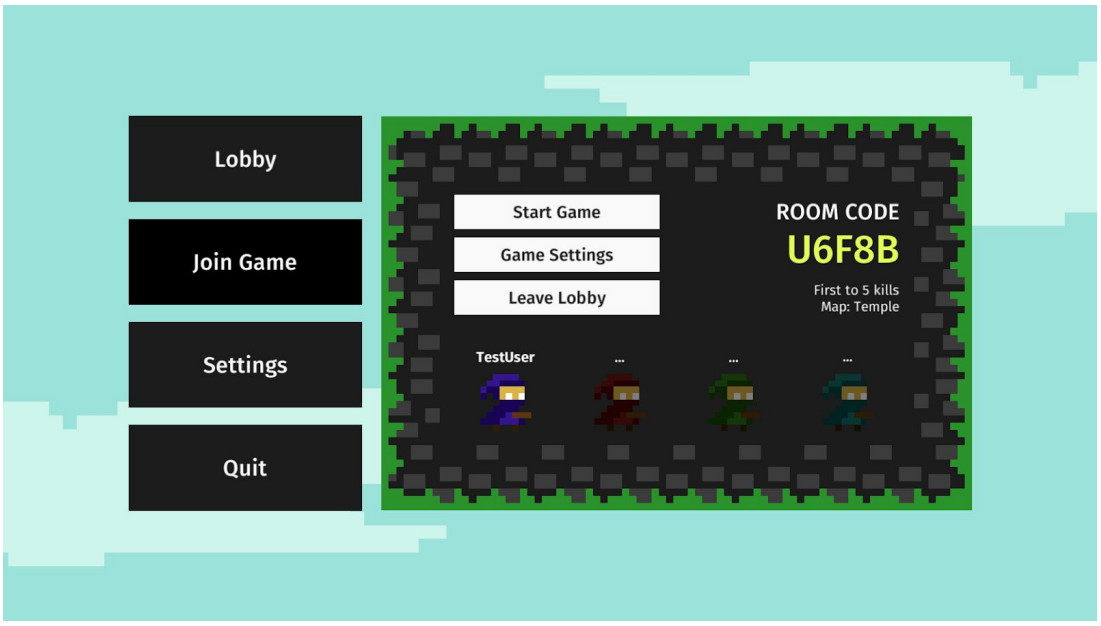
Lobby - opens the lobby screen.

The **Join Game** button is deactivated.

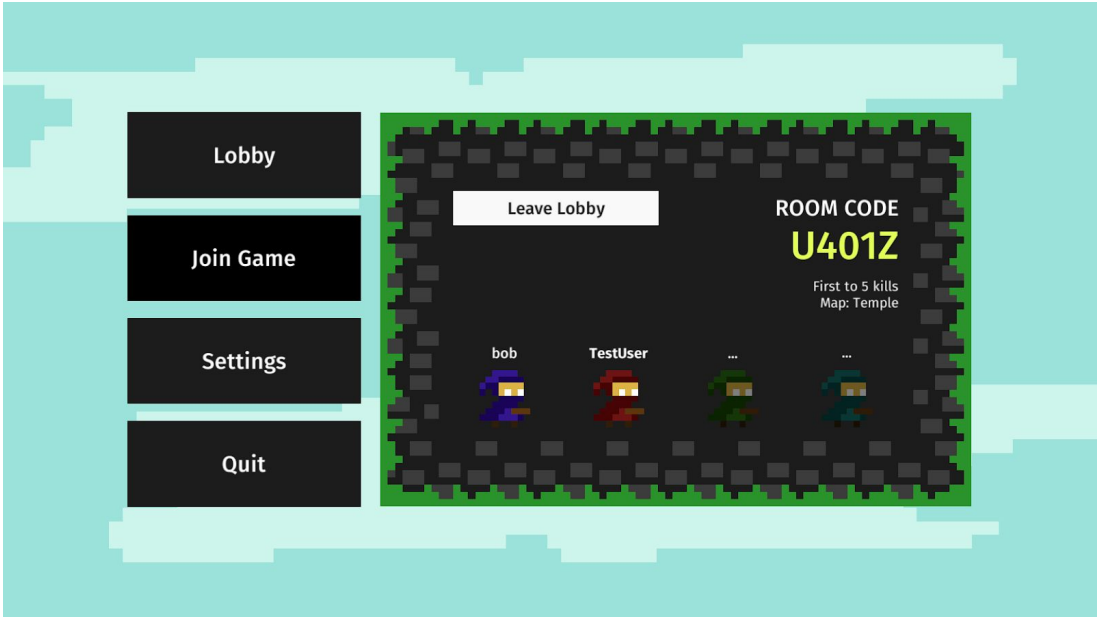
Screens

Lobby Screen *LobbyScreen.cs*
When the player creates or join a game, they are taken to the **Lobby** screen. Here, the players can wait for each other before starting the game. The host has the ability to change both the gamemode and map.

When in the lobby, the **Create Game** button is changed to **Lobby** button which redirects to the lobby screen. Also, the **Join Game** button is disabled.

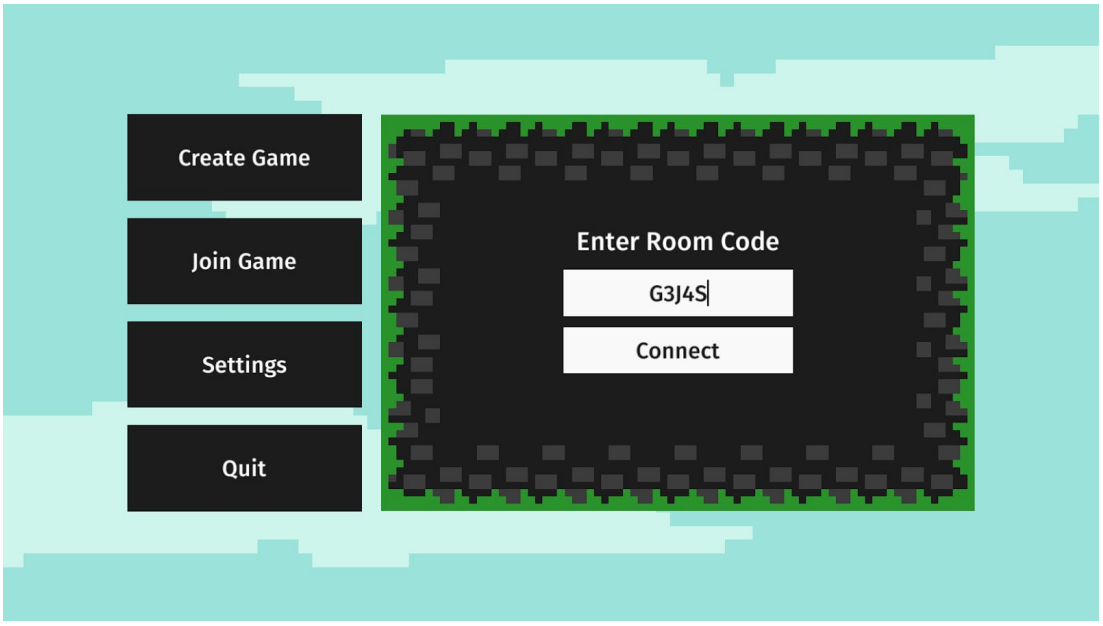


Here's what the lobby looks like for a player who's *not* the host.

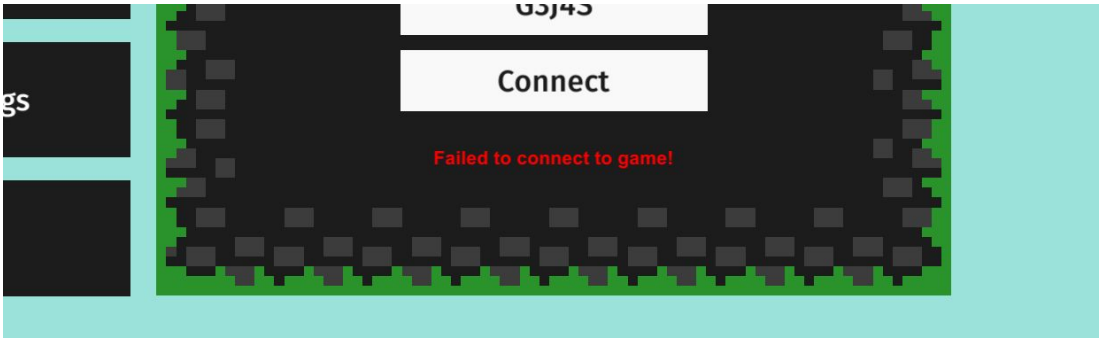


Connect Screen *ConnectScreen.cs*

When the player clicks on the **Join Game** button, they're taken to this screen. Here, they can enter in a room code and try to connect to a game.

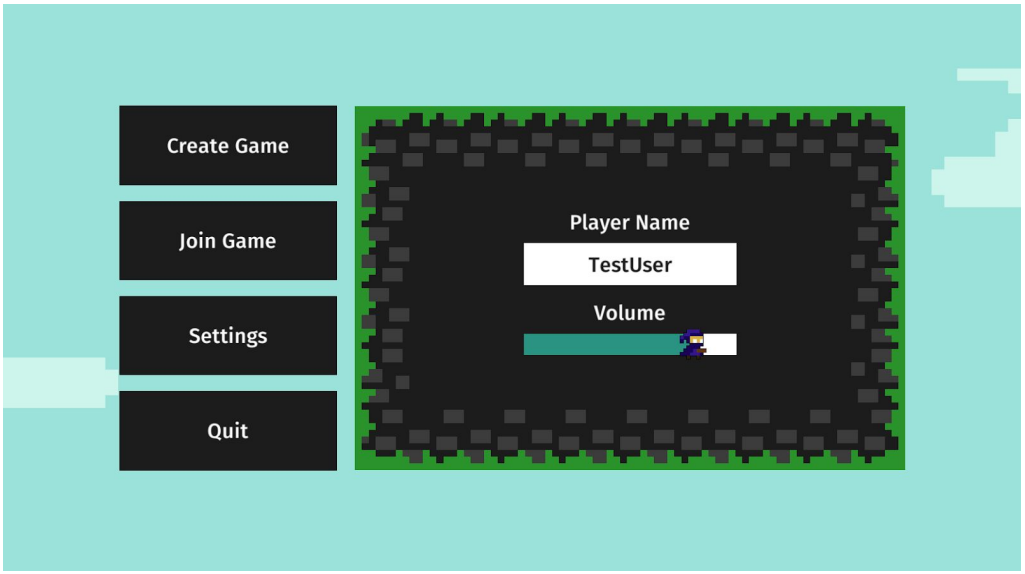


If the room does not exist, the following error will appear.



Settings Screen *SettingsScreen.cs*

When the player clicks on the **Settings** button, they're taken to this screen. Here, the player can change their name and the volume. Both the name and volume are saved to PlayerPrefs.

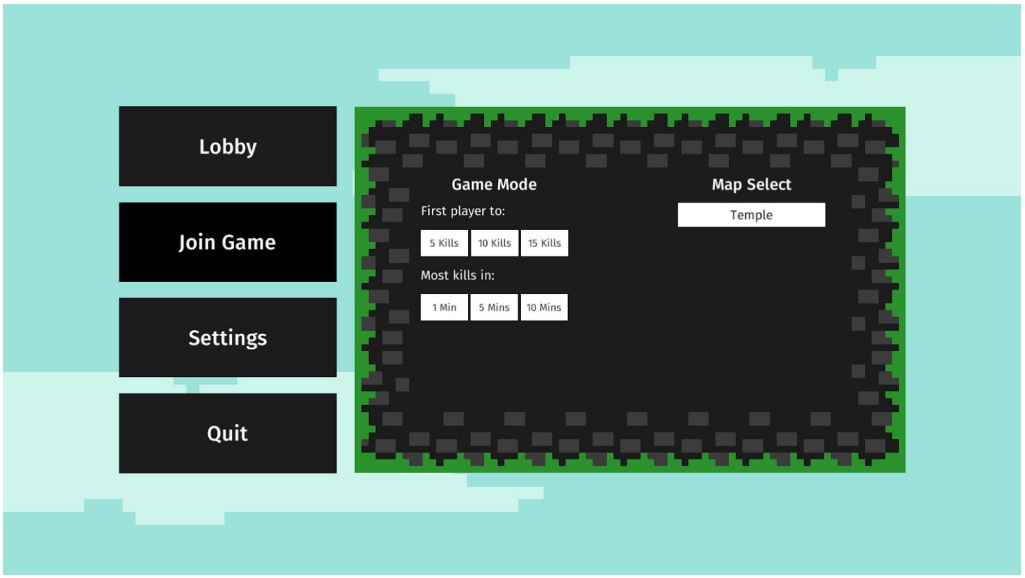


Game Settings Screen *GameSettingsScreen.cs*

If the player is the game host, on the **Lobby** screen they can click on the **Game Settings** button. Here, they can change the gamemode and map. These values are saved to the Photon room's CustomProperties.

Gamemode is saved with the key of **gamemode** and value of an int, referring to the *GameMode* enumerator. The value (how many kills, what duration) is saved with a key of **gamemodeprop** and a value of an int, referring to either the amount of kills or seconds.

Map is saved with the key of **map**, and a string value referring to the map name.



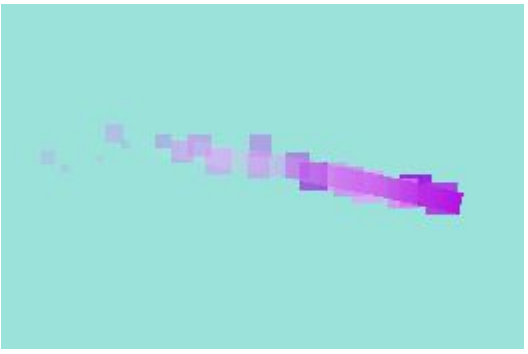
Relevant Scripts: *MenuUI, LobbyScreen, ConnectScreen, SettingsScreen, GameSettingsScreen.*

Spells

Spells are the way players fight each other in the game. There are **2** types of spells in the game.

- **Projectile** spells are objects that the player can shoot from their character. If the projectile hits another player, the damage and other properties are applied.

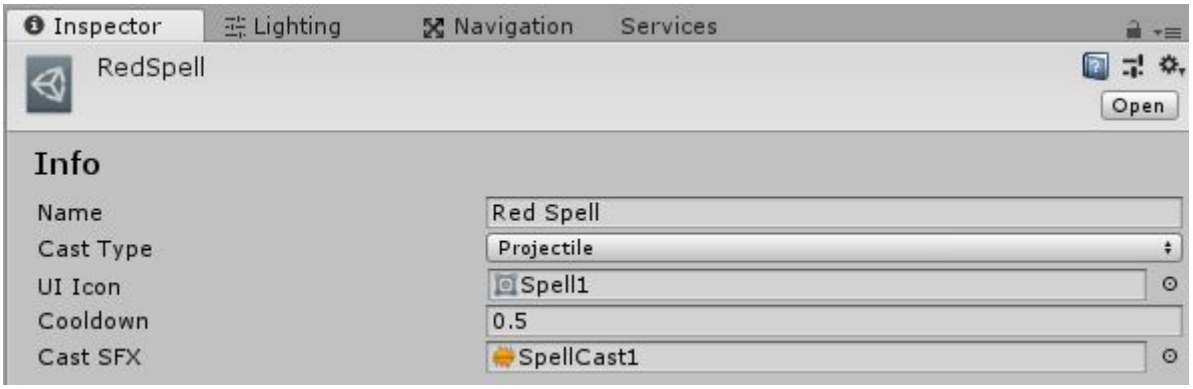
- **Self** spells are spells that the player casts upon themself. This can be to heal, teleport or damage nearby enemies.



Both types of spells have the base class of **Spell.cs**. This contains info such as the spell data, caster ID and a bool to check if this is *my* spell.

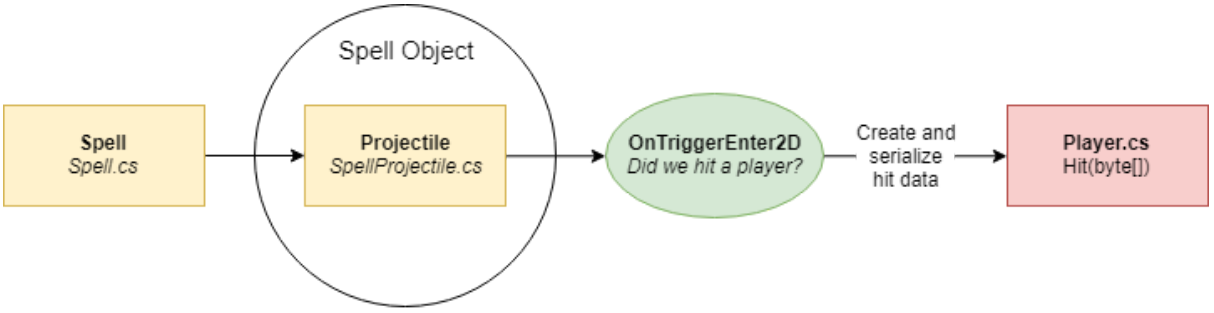
SpellData.cs is the class we use to contain the spell data, such as damage, speed, prefabs, etc. This is a *Scriptable Object*, so we can create them as assets in the editor and fill in the info in the **Inspector**.

- **Name** is the name of the spell (needs to be unique!).
- **Cast Type** is the type of spell (*Projectile* or *Self*).
- **UI Icon** is the sprite that's displayed on-screen.
- **Cooldown** is the min duration allowed between shots.
- **Cast SFX** is the sound effect that's played when the spell is cast.



Projectile Spells

When a projectile spell is shot, it checks for trigger enters with other players. If it hit another player, we then create a **SpellHitData.cs** class, serialize it and RPC the player's *Hit()* function. We're serializing the class because Photon has some issues sending over custom classes.

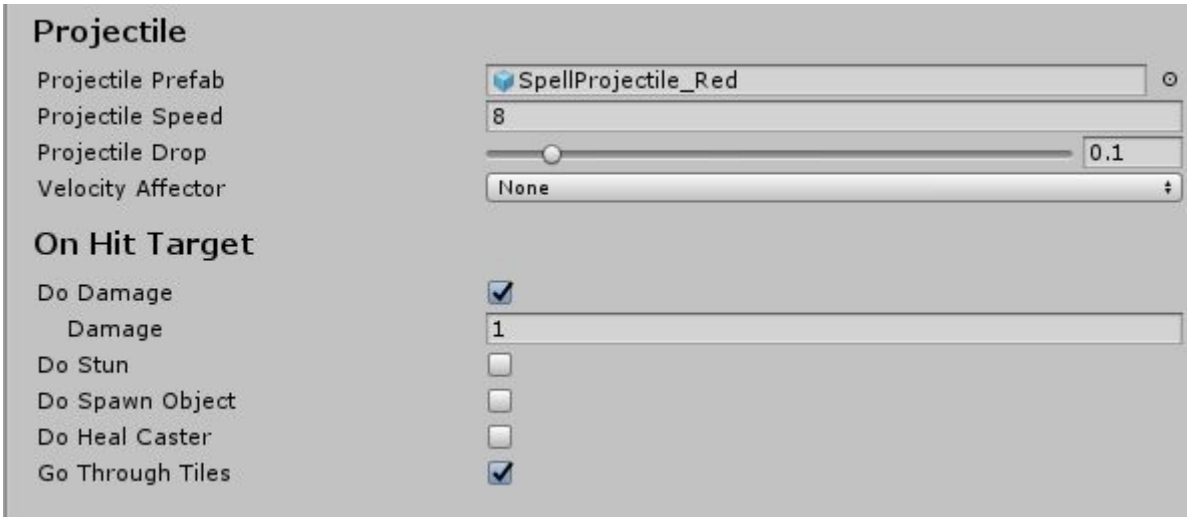


On the **SpellData** object, if the spell type is *Projectile*, the following properties will appear:

- **Projectile Prefab** is the object that's created when cast.
- **Projectile Speed** is the velocity given to the spell upon it being casted.
- **Projectile Drop** is the gravity given to the spell.
- **Velocity Affector** is a modifier you can apply to the spell's movement. E.g. make it move in a sine wave pattern.

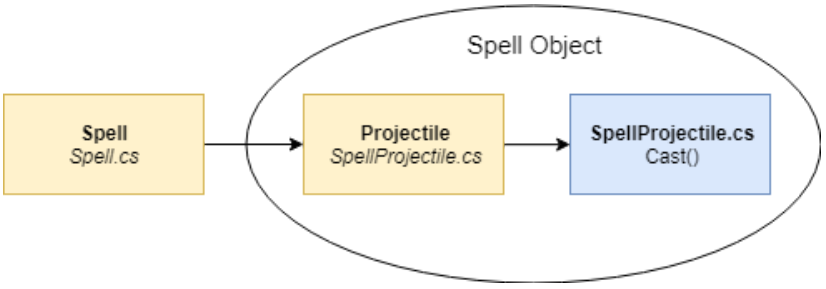
On Hit Target is what happens when the projectile hits an enemy player.

- **Damage** the hit player.
- **Stun** the hit player for a certain duration.
- **Spawn** and object at the hit player's position.
- **Heal** the caster of the spell a certain amount.
- And make the spell be able to **go through tiles**.



Self Spells

When a self spell is cast, we create an empty GameObject and add the **SpellSelf.cs** script to it. Then we transfer over the initial data for it to work with. This script then goes through the SpellData and does whatever it needs to. Once these actions have taken place, the empty GameObject is destroyed.



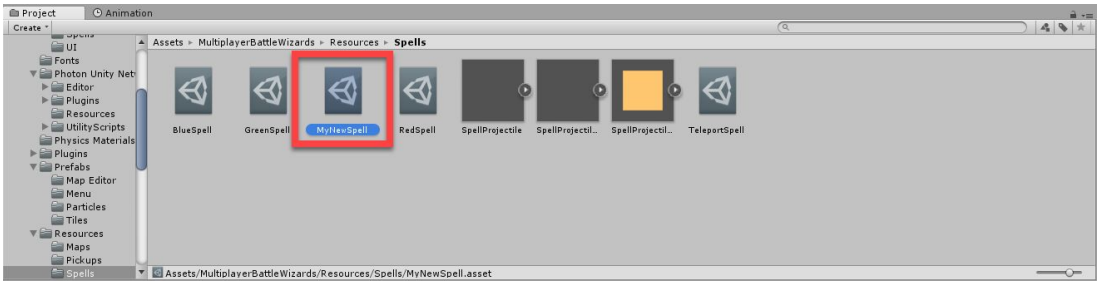
On Self Cast is what happens when the player self casts this spell.

- **Heal** the caster a certain amount.
- **Damage** nearby enemies.
- **Stun** nearby enemies.
- **Teleport** to either: the furthest player, nearest, random, random position or top of the map.
- **Spawn** an object at the caster's position.

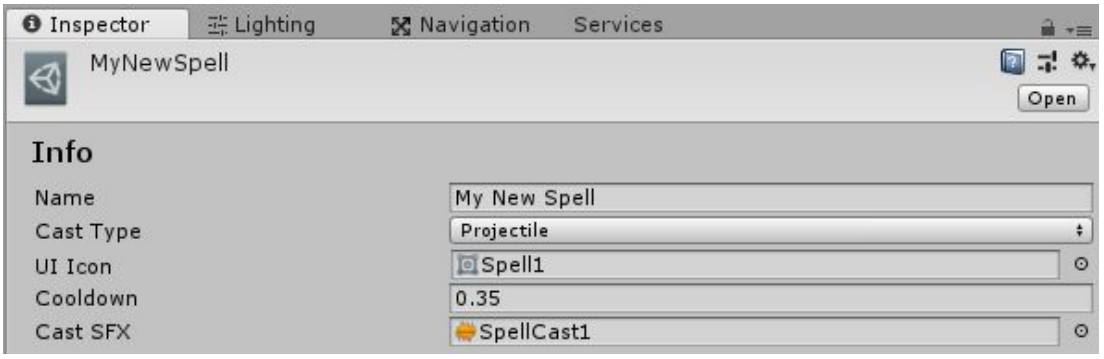
How to Create a Spell

Creating spells is fairly straightforward, with no scripting required.

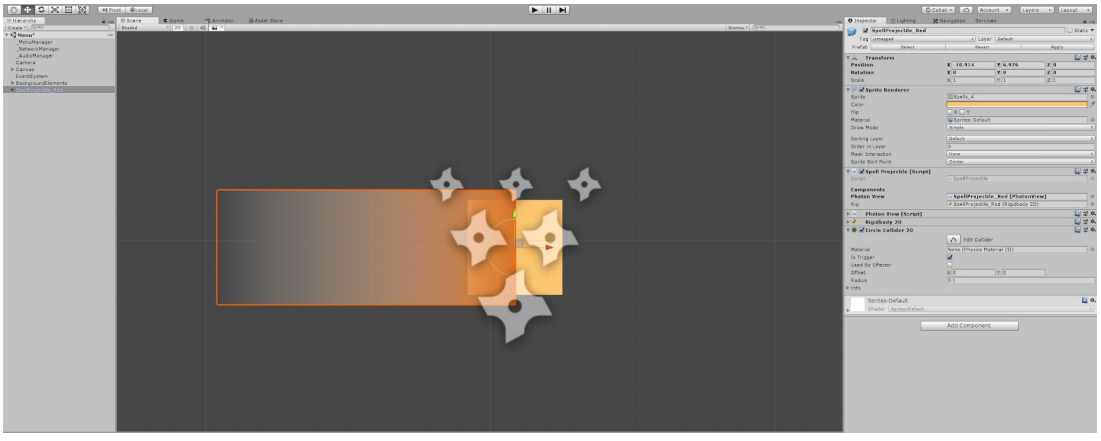
1. Navigate to the *Resources/Spells* folder and create a new **SpellData** asset (right click Project > *Create* > *New Spell*). You can call this whatever you want.



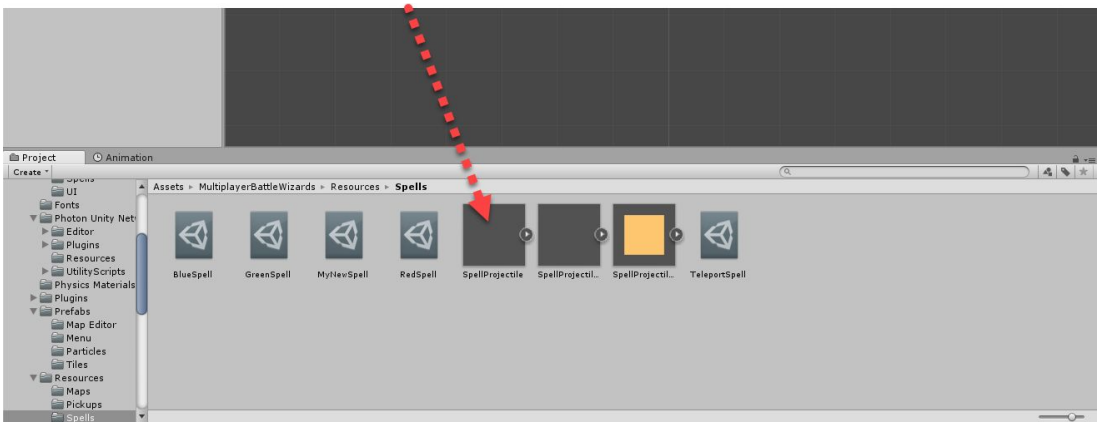
2. Selecting it, we can start to fill in the properties over at the Inspector.
    a. Make sure the name is *unique* as this is how they're transferred across the network when referencing a spell.



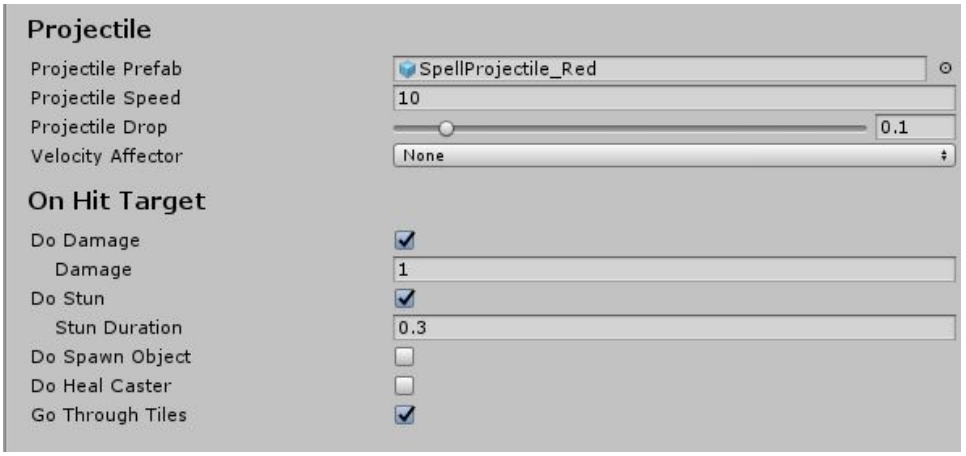
3. If you're making a **projectile** spell...
    a. Drag in an existing spell projectile prefab or the template (*Prefabs* folder) and create the visual look.



b. When that's done, drag the spell prefab into the *Resources/Spells* folder.

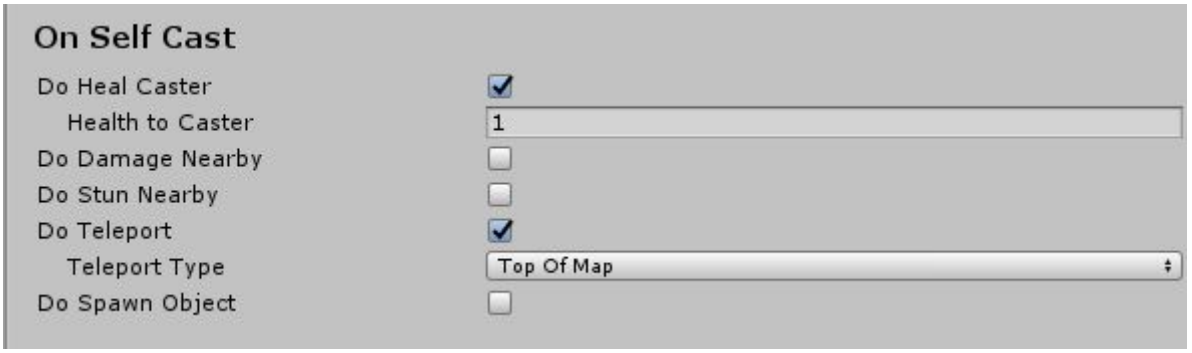


c. Back in our **SpellData** asset, we can fill in the rest of the details.

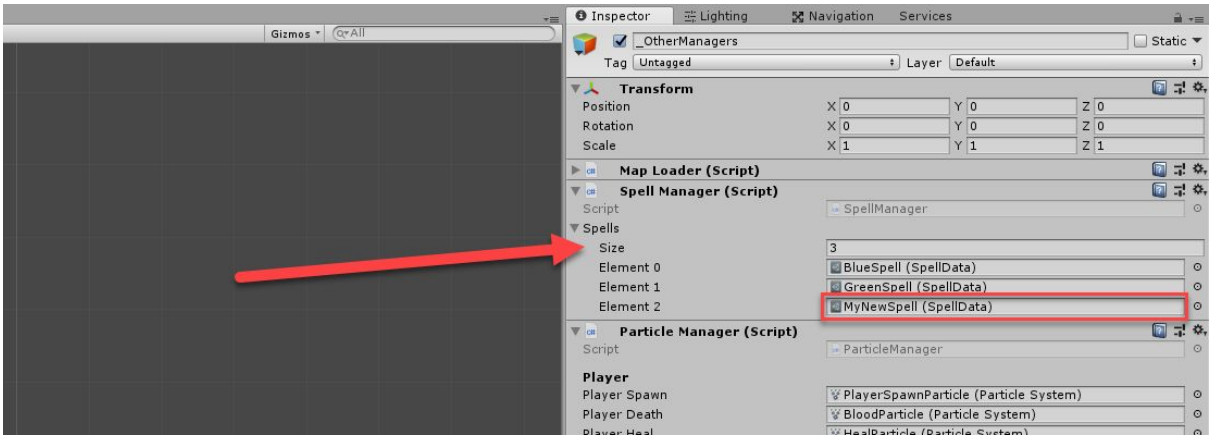


4. If your making a **self** spell…

a. Fill in the rest of the info in the **SpellData** asset.



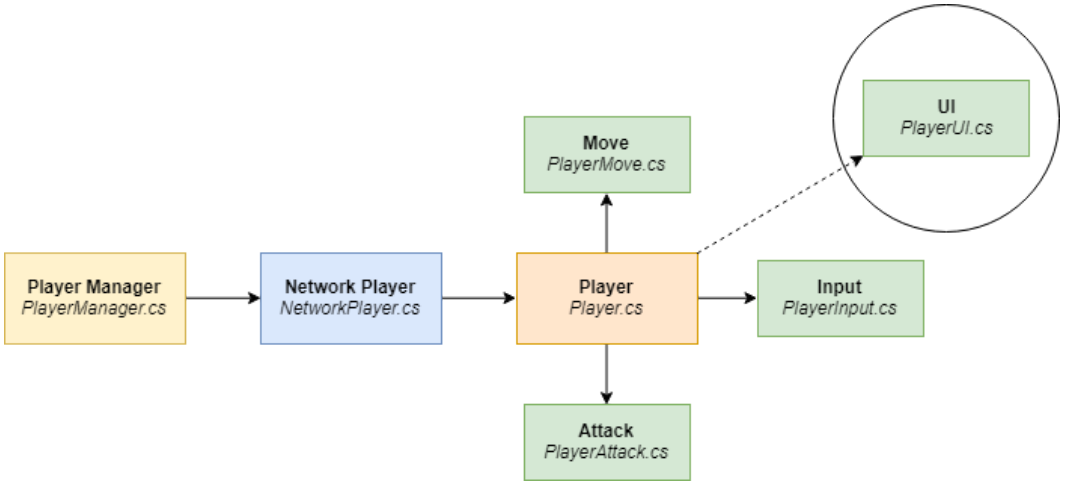
5. Next, let's go over to the **Game** scene. Select the **_OtherManagers** object and drag your new spell into the **Spell Manager**'s spells list.



6. If you want to be able to get this spell from a pickup, you'll have to make that too. Learn how here.

Players

All the players are managed by the **Player Manager**. Each player connected to the game and playing are a **Network Player**. The network player has the in-game representation of the player which is **Player**. The player then has various components that manage different functions like: moving, attacking, input and UI. Here's a hierarchy of how the players are connected to their components and manager.



Player Manager *PlayerManager.cs*
Contains and manages all the players in the game. Can fetch and return players by ID, GameObject, etc. Initializes and spawns in the players at the start of the game.

Network Player *NetworkPlayer.cs*
The "top layer" for the player. Contains their network ID, corresponding PhotonPlayer object and Player object.

Player *Player.cs*
The core script for each player. Manages health, taking damage, spawning, death, states, connecting the following components, etc.

Move *PlayerMove.cs*
Moves the player around in world space. Also in charge of jumping and ground detection.

Attack *PlayerAttack.cs*
Shoots spells from the player and manages their current spell.

Input *PlayerInput.cs*
Checks for local input and runs events that the other components can listen for.

UI *PlayerUI.cs*
Manages the player's UI heart icons and score. Only one of the 4 components not attached to player object.

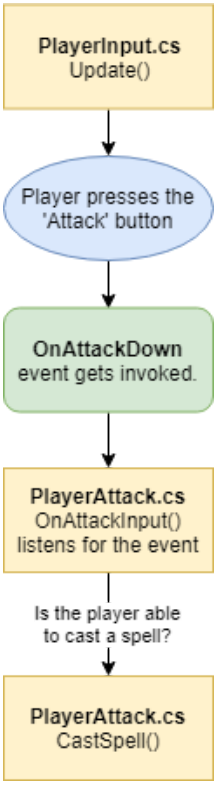
States

The player has **5** states that they will change through during the game. These states are also tied to its Animator components, so the respective animations will also play. Every frame, the state of the player is checked in their *SetState()* function.

- **Idle** when the player is standing still. This is the default state.
- **Moving** when the player is moving horizontally, but not vertically.
- **InAir** when the player is moving vertically.
- **Stunned** when the player is stunned and cannot move or attack.
- **Dead** when the player has died and is off-screen.

Casting Spells

Each player can casts spells. These are how you affect both other players and yourself in the game to win.

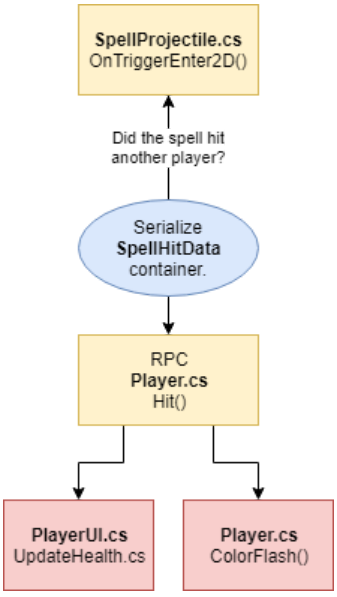


1. In the *Update* function off the **PlayerInput** class, we check for the 'Attack' button to be pressed down.
2. Then we invoke the *OnAttackDown* event.
3. The *OnAttackInput* function in the **PlayerAttack** class is listening for this, it will then check if the player is able to cast a spell and if so - call the *CastSpell* function.
4. This instantiates the spell for that client only.
5. Then for all other clients, we call the *CastSpell* function again through an RPC to create it for them.

We do it this way to have a spell that is exactly the same across all clients, even the one who casted it. When the spell hits an enemy on the caster's screen, it does the same for all others. We *don't* sync the position or instantiate it through Photon, as it's movement is very predictable.

Taking Damage

This game has client side hit detection. So when the caster of a spell sees it hit another player on their screen, it sends the RPC.



1. If the spell projectile enters the trigger of a player, we check if it's caster's projectile, or just the visual for the other clients.
    a. If it's **not** the caster, just destroy it.
2. If it **is** the caster, create a **SpellHitData** class to hold info like attacker id, the spell, hit direction, etc.
3. Serialize that as a **byte array** as Photon has trouble sending over custom classes.
4. RPC it to the hit player's **Hit** method.
5. There, we deserialize the byte array and apply any damage, effects, etc to the hit player across all clients.
6. Then the hit player updates their health UI and flashes their sprite red.

Death

If the player's health reaches 0, we call the *Die* function locally first. This is so the effects of doing so are immediately seen on our screen. Then we RPC the *Die* function again, so it will be updated on the other player's screens.

When a player dies, there's a few things that happen:

Locally

- Disable physics.
- Set position to be outside of the map.
- RPC the *Die* function for all **other** players.
- If we had a current attacker, RPC their *KillPlayer* function.

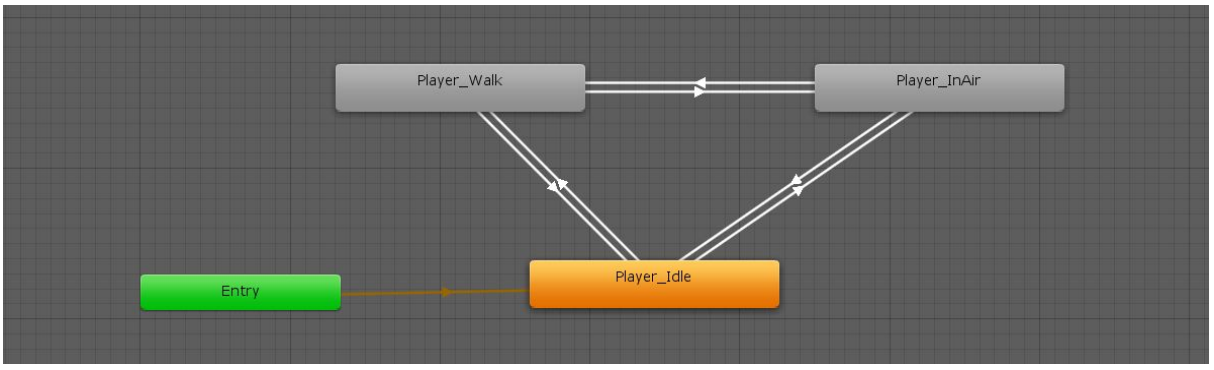
Globally

- Set health to 0 and state to Dead.

Animations

By default, there are **4** different colored players (purple, red, green and blue). Each has their own Animator Controller. The animations reflect the player's current state and is connected by an *int* parameter called **State**. The player has 3 different animations.

- **Idle** animation is played when the player is in state *Idle*.
- **Walk** animation is played when the player is in state *Moving*.
- **InAir** animation is played when the player is in state *InAir*.

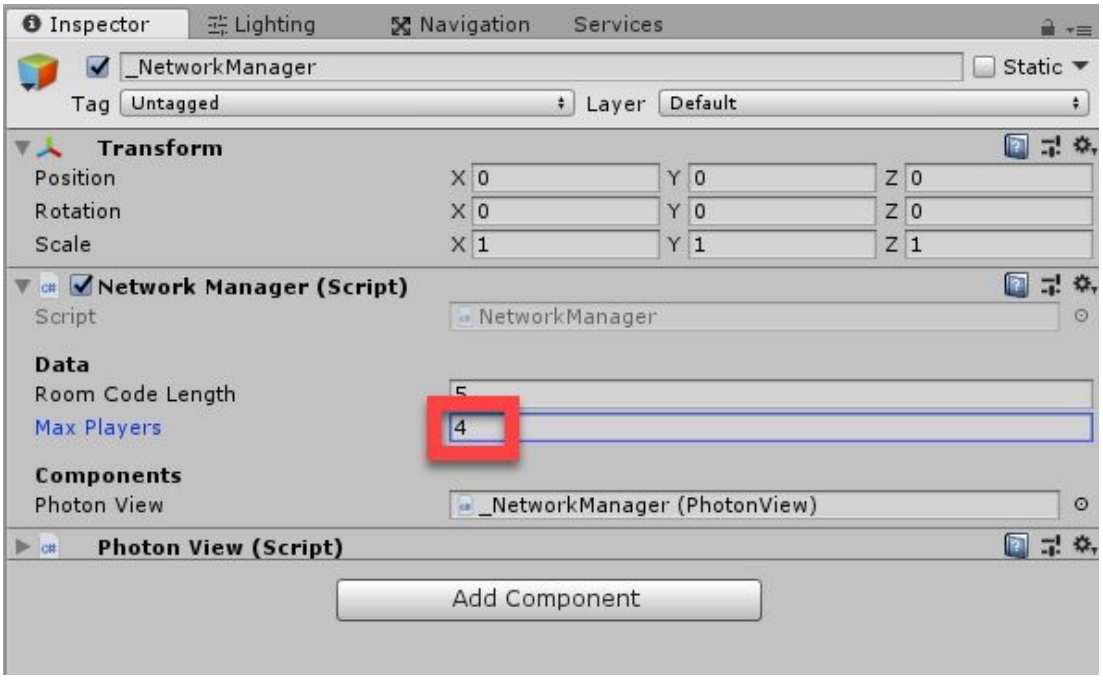


The player animations and controllers are located in the *Animations/Player* folder.

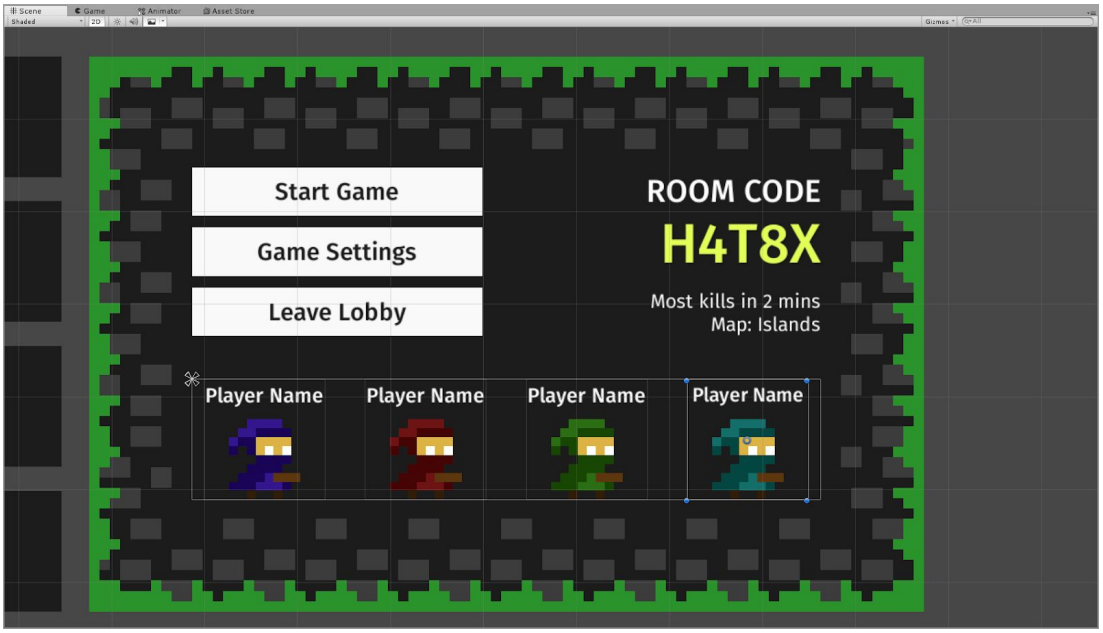
How to Add More Players

Let's say you want more than the 4 default players.

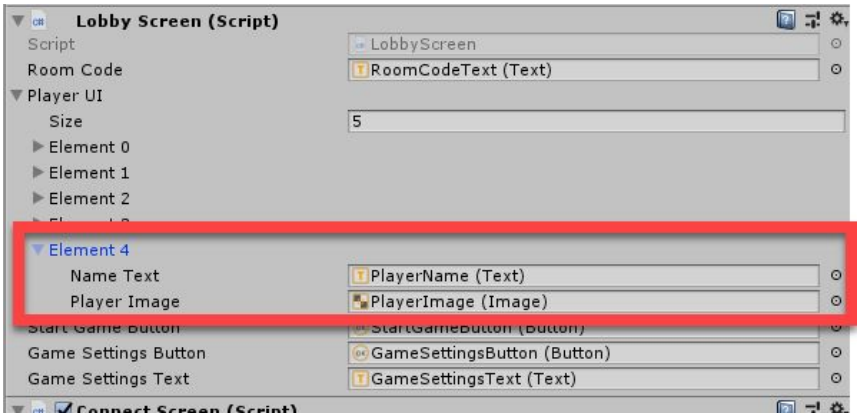
1. First, go to the **Menu** scene and change the NetworkManager's *Max Players* to your new player count.



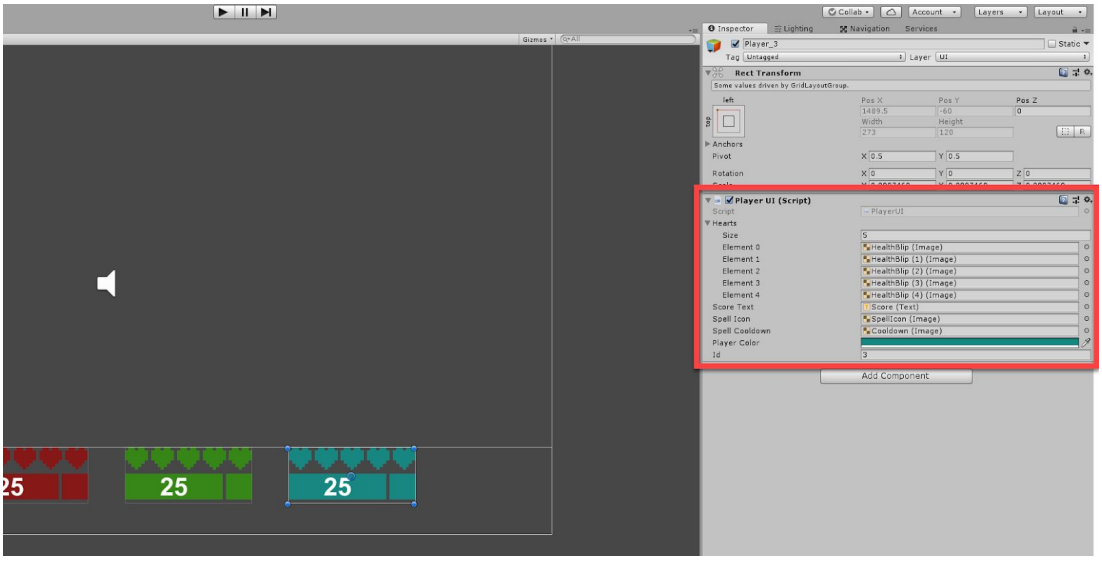
2. Next, you want to enable the **LobbyScreen** object (*Canvas > Screens > LobbyScreen*) and add a new player to the *Players* grid layout.



3. Then select the **_MenuManager** object and add your new UI player text and image to the **Lobby Screen** component's *Player UI* array.

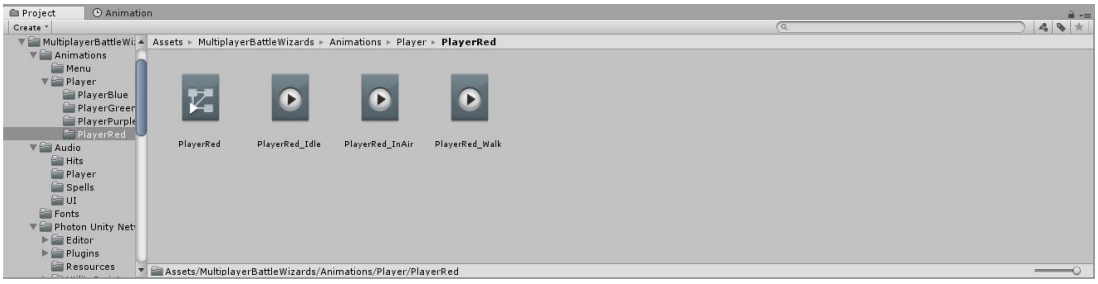


4. Let's now go over to the **Game** scene. Duplicate one of the existing player UI objects, change the color and ID.



5. Then select the **_OtherManagers** object and add your new player UI object to the *PlayerUIs* array.

6. Finally, you need to duplicate an existing player animations folder (*Animations > Player > [any player folder]*) and add in your corresponding sprites.



7. After that, just select the **_GameManager** object and drag in your new Animator controller to the **PlayerManager**'s *Player Animators* array.

Relevant Scripts: *PlayerManager, NetworkPlayer, Player, PlayerMove, PlayerAttack, PlayerInput, PlayerUI.*

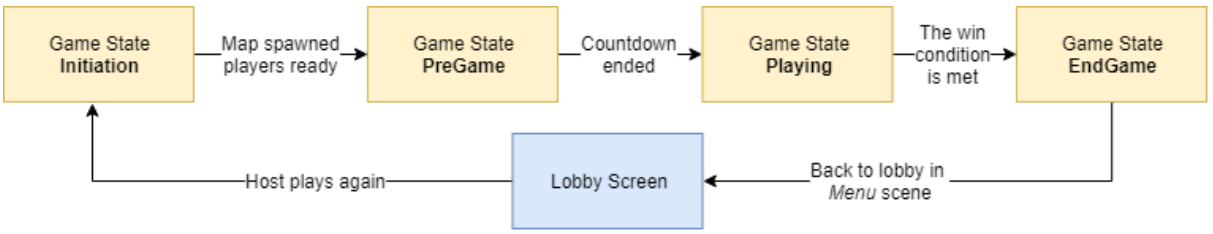
Game Manager

The **Game Manager** (*GameManager.cs*) is the script that manages the core aspects of the game in the *Game* scene.

States

There are **4** game states that keep track of the current state of the game. These dictate what gets checked and what things are allowed to happen.

- **Initiation** - if we're currently waiting for players, spawning the map, initiating players.
- **PreGame** - counting down before the game begins.
- **Playing** - when the game is in progress.
- **EndGame** - when the game has ended and a player has won.



Gamemode

There are **2** gamemodes in the game currently. These dictate the win condition for the game.

- **ScoreBased** - first player to specific score (kills) wins.
- **TimeBased** - after a set duration, the player with the most kills wins.

If the gamemode is **score based**, we check each time a player has been killed - is the killer's score equal to the winning score? If so - they win the game.

If the gamemode is **time based**, we check each frame - has the current time met the time limit? If so, get the player with the highest score and they win the game.

Spell Distribution

There are **2** ways that spells are distributed to players - *not* including through pickups.

- **RandomOnStart** - random spell is given to the player at the start of the game.
- **RandomOnSpawn** - random spell is given every time the player spawns.

Managers

There are a few scripts which act as *managers*. These are scripts that manage a certain aspect of the game - players, particles, networking, etc.

Network Manager *NetworkManager.cs*

Takes care of connecting to Photon, creating / joining rooms, loading scenes, getting room codes, player lists and many server related functions.
This is also the only script that transitions between the *Menu* and *Game* scene.

Player Manager *PlayerManager.cs*

Manages all the players in the game. Initiating and spawning them in. It's also used for finding players by their id, object, etc.

Game Manager *GameManager.cs*

Runs and manages the core game. Game mode, timing, checks for win condition, starts and ends the game. Also contains some constant variables that all players can access like the height at which they'll die if they fall off the map.

Map Loader *MapLoader.cs*

Spawns in the map. Deserializes the map's JSON file to a *MapData* object, then instantiates each of the tiles and sets the spawn points.

Spell Manager *SpellManager.cs*

Holds all the spells in the game and can return random spells when asked.

Pickup Manager *PickupManager.cs*

Spawns pickups over time.

Particle Manager *ParticleManager.cs*

Manages the spawning and destroying of particles.

Audio Manager *AudioManager.cs*

Manages the playing and stopping of audio clips.

Menu UI *MenuUI.cs*

Takes care of changing pages, toggling buttons, etc for the *Menu* scene.

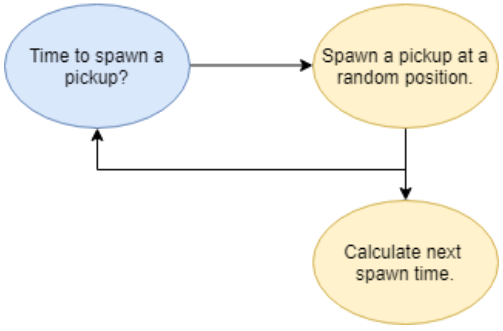
Relevant Scripts: *NetworkManager, PlayerManager, GameManager, MapLoader, SpellManager, PickupManager, ParticleManager, AudioManager, MenuUI.*

Pickups

The game features a basic pickup system. There are *two* types of pickups:

- **Health** pickups, give the player a set number of health points upon picking it up.
- **Spell** pickups can either give the player a specific or random spell upon picking it up.

The **PickupManager.cs** manages and spawns the pickups over time. The checking and spawning is done by the host.



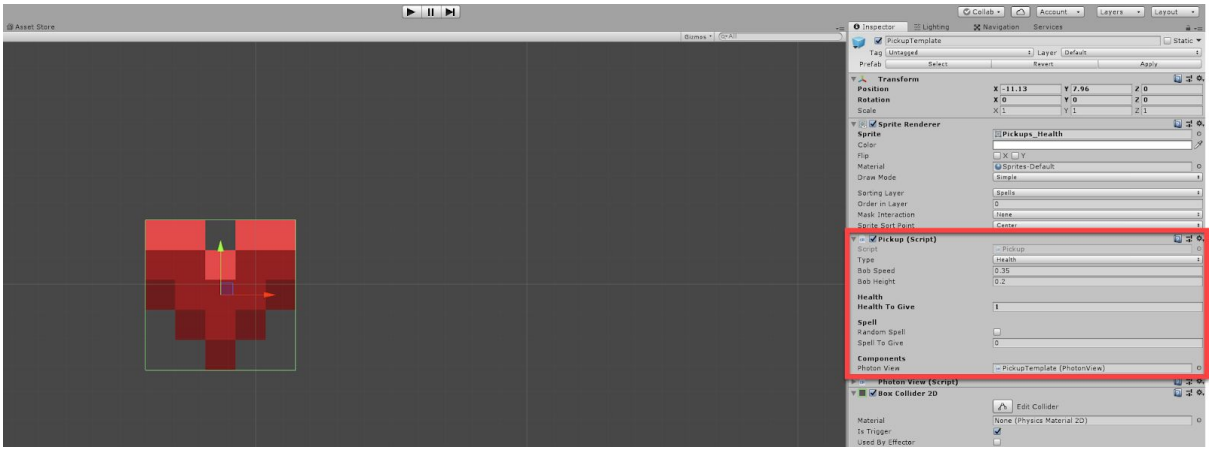
1. Host calculates a *next time to spawn pickup* (*GetNextSpawnTime*).
2. When it's that time:
    a. Spawn the random pickup (*SpawnPickup*).
3. Repeat.

When a pickup is spawned, detecting when a player enters it is also done by the host. This is to prevent multiple players from picking it up at the same time.

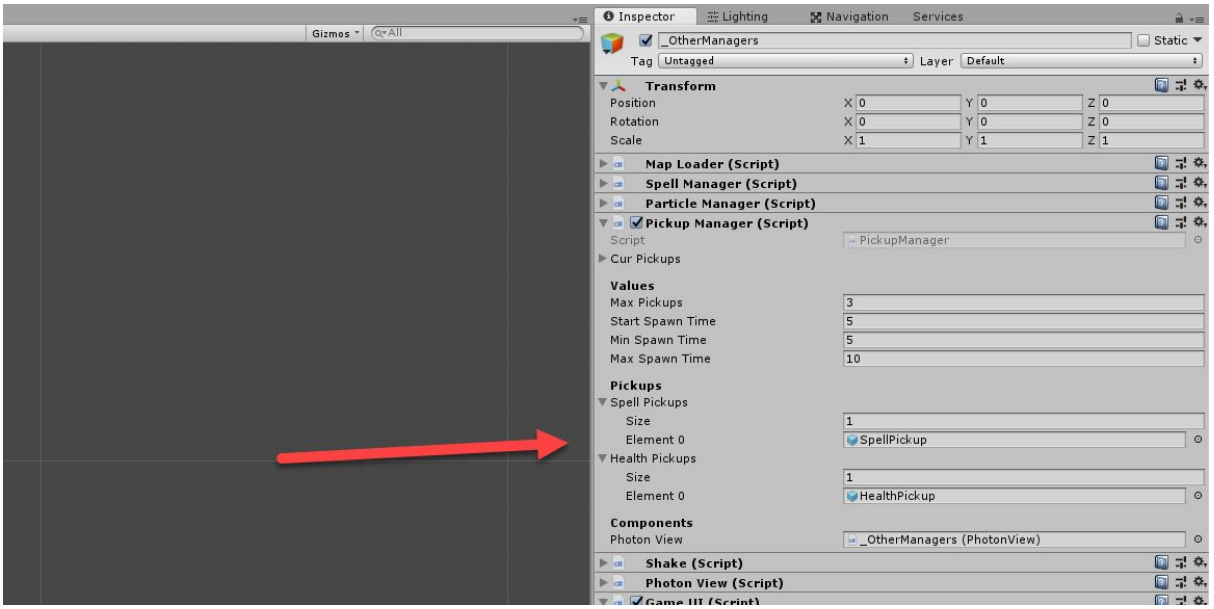
How to Create a Pickup

1. Drag the **PickupTemplate** object (*Prefabs* folder) into the scene (or any existing prefab you wish you make a copy of).

2. Fill in the details in the **Pickup** component and then save your pickup as a prefab in the *Resources/Pickups* folder.



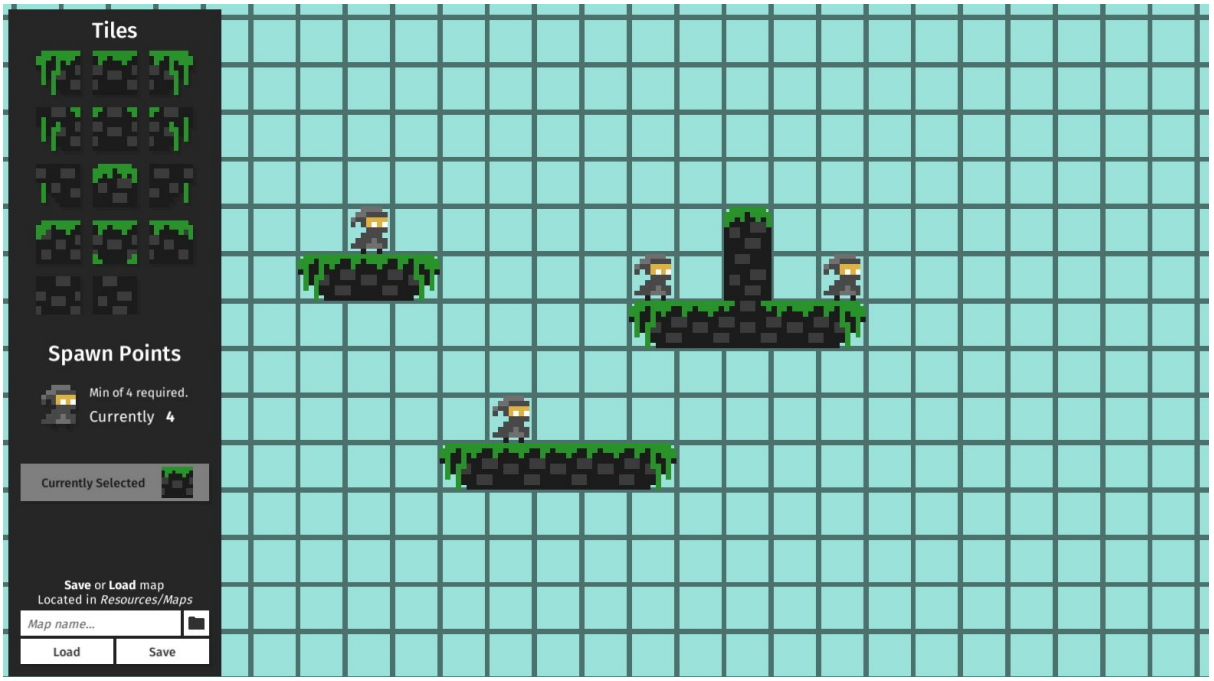
3. Then in the **Game** scene, select the **_OtherManagers** object and drag the pickup prefab into either the **Spell Pickups** or the **Health Pickups**, depending on which type it is.



Relevant Scripts: PickupManager, Pickup.

Map Editor

There is an included **map editor** (*MapEditor.unity* scene) which allows you to create and modify maps for the game.



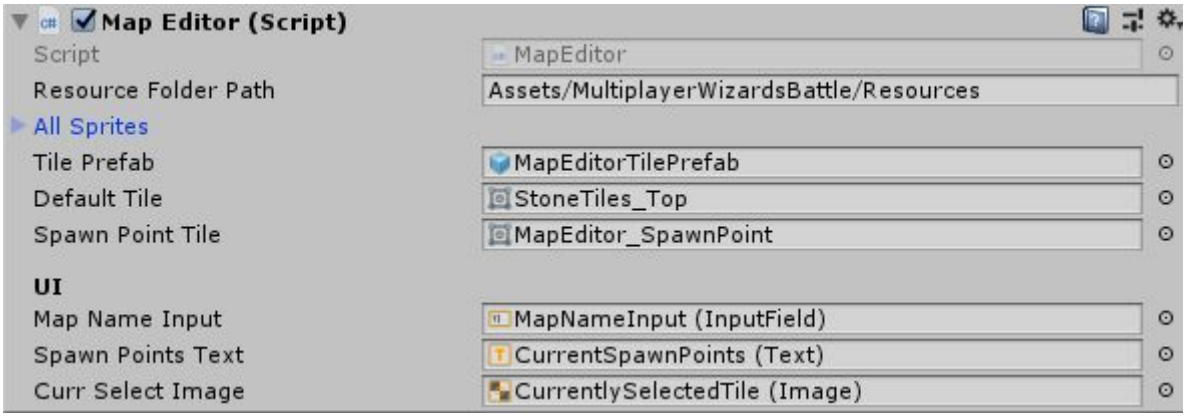
To start editing maps, you must be running the **MapEditor** scene inside of the Unity Editor. This will not work in build version so don't include the scene in the build settings.

- Maps are saved as .json files, which are serialized versions of the **MapData** class.
- Maps are saved in the *Resources/Maps* folder.
    - By default, this is located: *Assets/MultiplayerBattleWizards/Resources/Maps*
    - If you don't have a *Maps* folder in Resources, create it.

- When you've made a map, enter in a name for it (make sure it's unique) and click on the **Save** button.
- If you want to load a map, enter in the map name and click on the **Load** button.
- Clicking the folder button takes you to the *Resources/Maps* folder.

The script that runs the map editor is **MapEditor.cs** (attached to the **_MapEditor** object).

Resource Folder Path	The file path to the *Resources* folder.
All Sprites	A list of all the sprites available to use in the Editor.
Tile Prefab	The base tile prefab.
Default Tile	The sprite for the tile you want as the default one.
Spawn Point Tile	Sprite to use as a spawn point tile.



There are a few requirements when saving a map:

- You need a min of 4 spawn points on the map (more the better).
- You need a name for the map (if it's the same as an existing one, it will replace it).

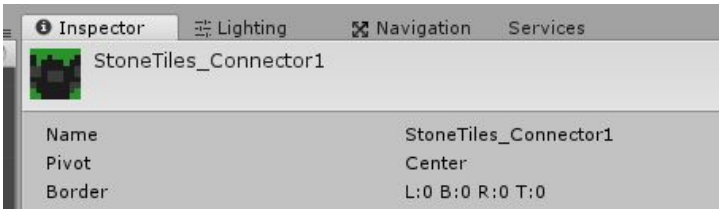
Editor Controls

Here are the keyboard and mouse controls for the editor:

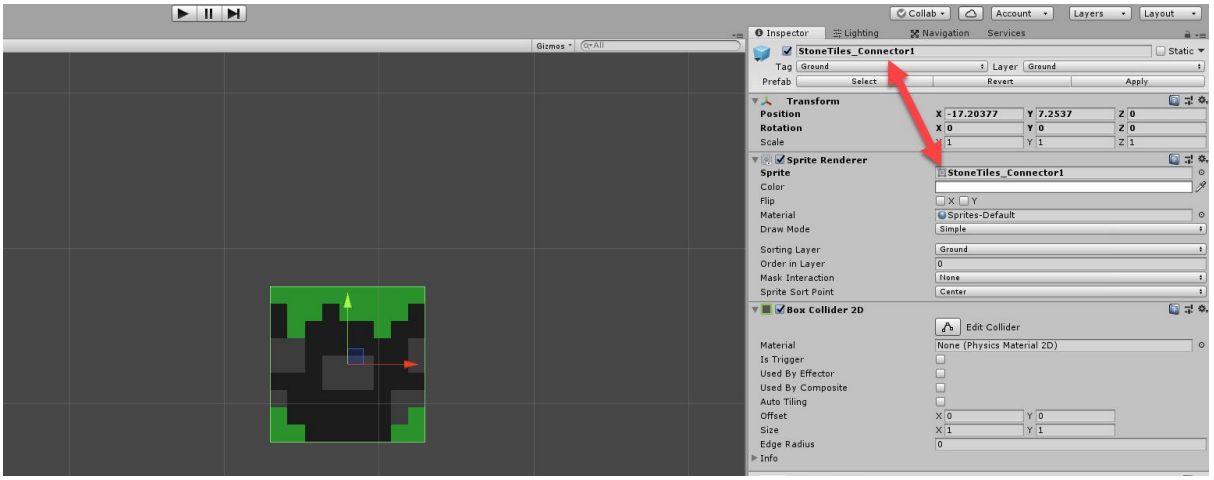
WASD	Move the camera around the scene.
Middle Mouse	Pan the camera around the scene.
Scroll Wheel	Zoom the camera in and out.
Left Mouse	Place a tile.
Right Mouse	Remove a tile.
F	Center camera position and size (how it will appear in-game).

How to Add a Tile

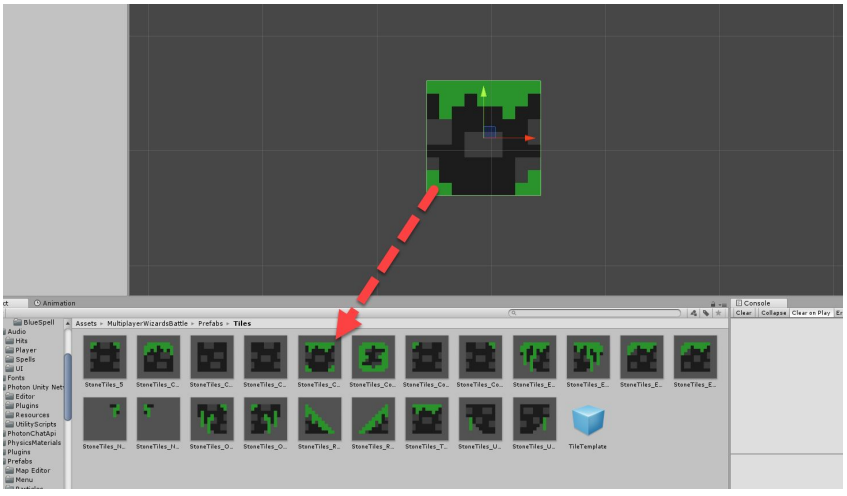
1. Import the sprite (or spritesheet) that you want to use as the tile. Make sure that the sprite has a unique name as that's how they are identified and saved to the map file.
    a. If you ever rename a sprite - the existing maps using that sprite will not work!



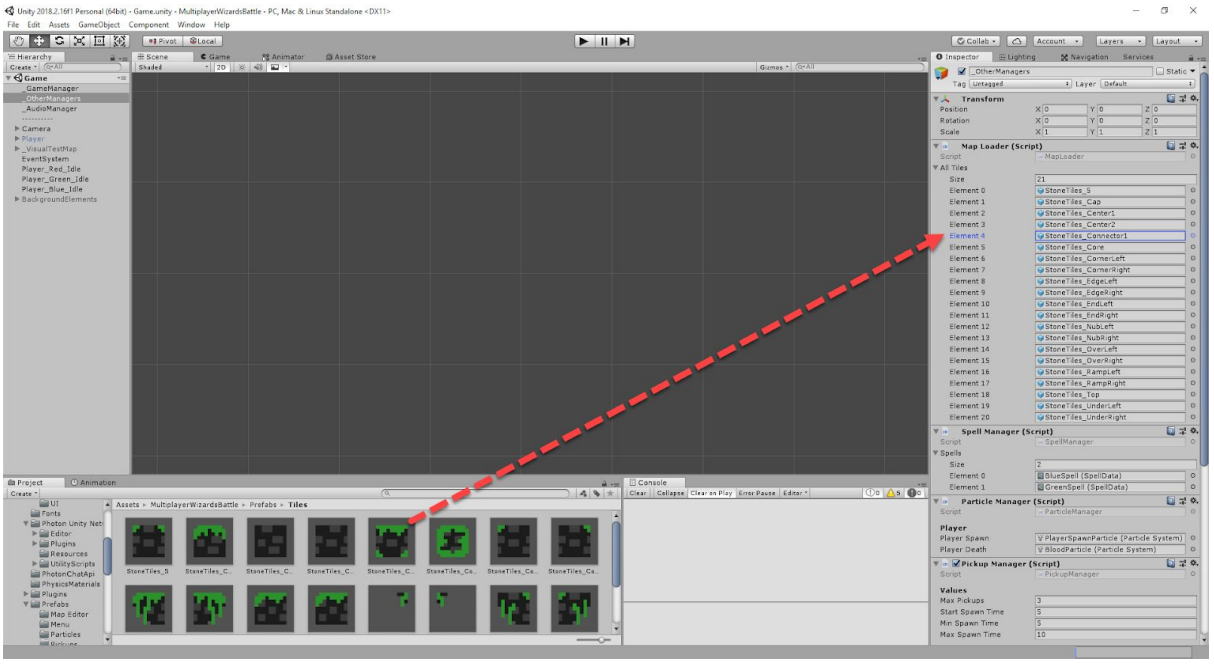
2. Now drag in the **TileTemplate** prefab (*Prefabs* folder).
    a. Add your sprite and make sure the collider fits.
    b. Change the name of the object to the sprite name (has to be exactly the same!).



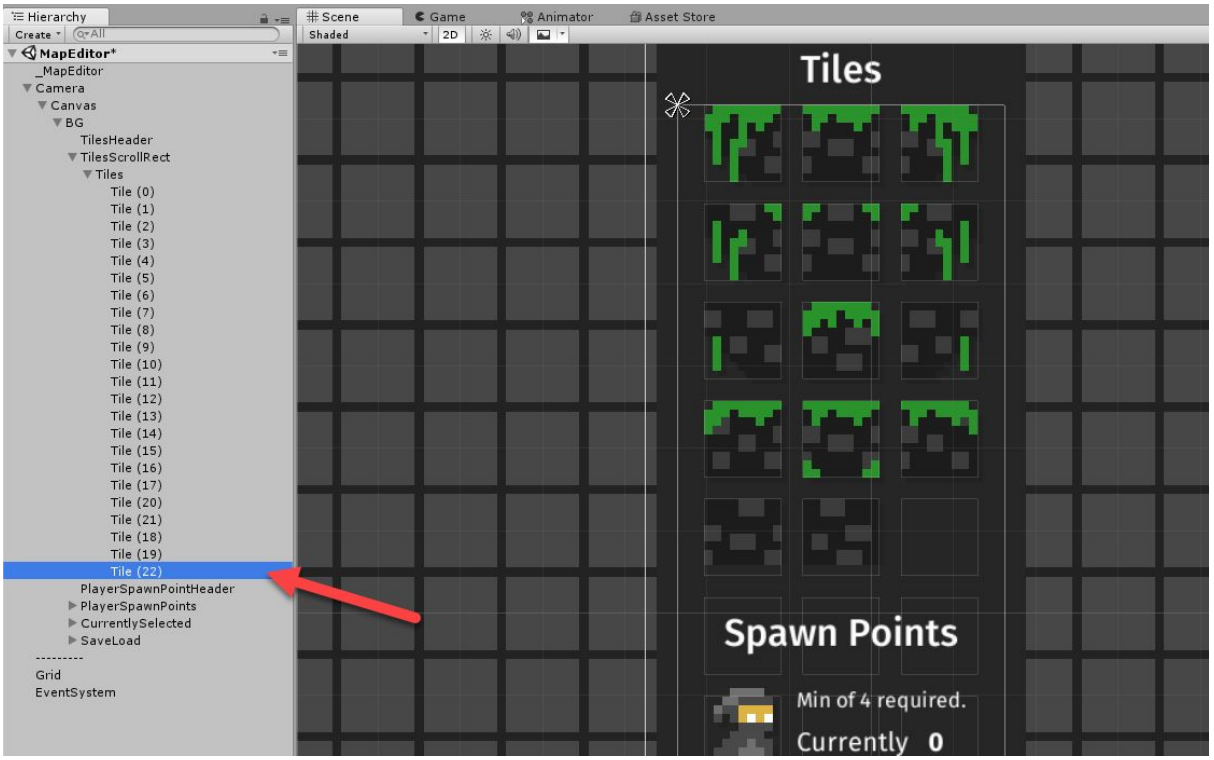
3. Drag the new tile object into the *Prefabs > Tiles* folder.

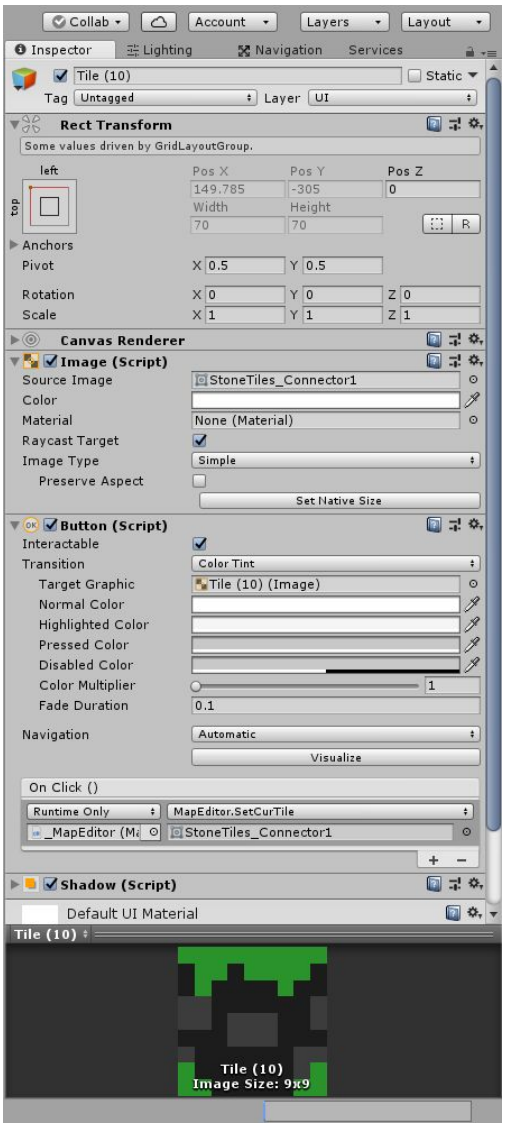


4. Now go to the **Game** scene and select the **_OtherManagers** object where we have the **MapLoader** script attached and drag your new tile object into the **All Tiles** list.



5. In the **MapEditor** scene, open up the tiles container and duplicate one of the tile buttons in the list.





6. Selecting the new tile, in the **Inspector**, we need to change a few things.

- Set the **Source Image** to be your new sprite.
- In the **OnClick** event, change the sprite to your new one.

You can now use your new tile in the *Map Editor* and in game!

Relevant Scripts: *MapEditor, MapEditorCamera, MapData.*

Events

GameManager

onMapLoaded - Called when the map has been loaded in.

- *PickupManager.OnMapLoaded*
    - Get the spawn points for pickups.

- *GameManager.OnMapLoaded*
    - Tell everyone we're in the game and ready to go.

onPlayersReady - Called when all the players are spawned in and ready to go.

- *GameUI.OnPlayersReady*
    - Begin countdown.

onGameStart - Called when the game begins and the players can start fighting.

- *Player.OnGameStart*
    - Player can now move and attack.

onGameWin<int> - Called when a player has won the game (int = winning player ID).

- *Player.OnGameWin*
    - Player can no longer attack.

PlayerManager

onPlayerInitialized<int> - Called when a player is initialized and spawned (int = player's ID).

- *PlayerUI.OnPlayerInitialized*
    - Connects the UI to the corresponding player.

You may notice that all of the events requiring a parameter are using **System.Action** rather than **UnityEvent**. This is because problems occured when trying to use the Unity event system with a parameter.

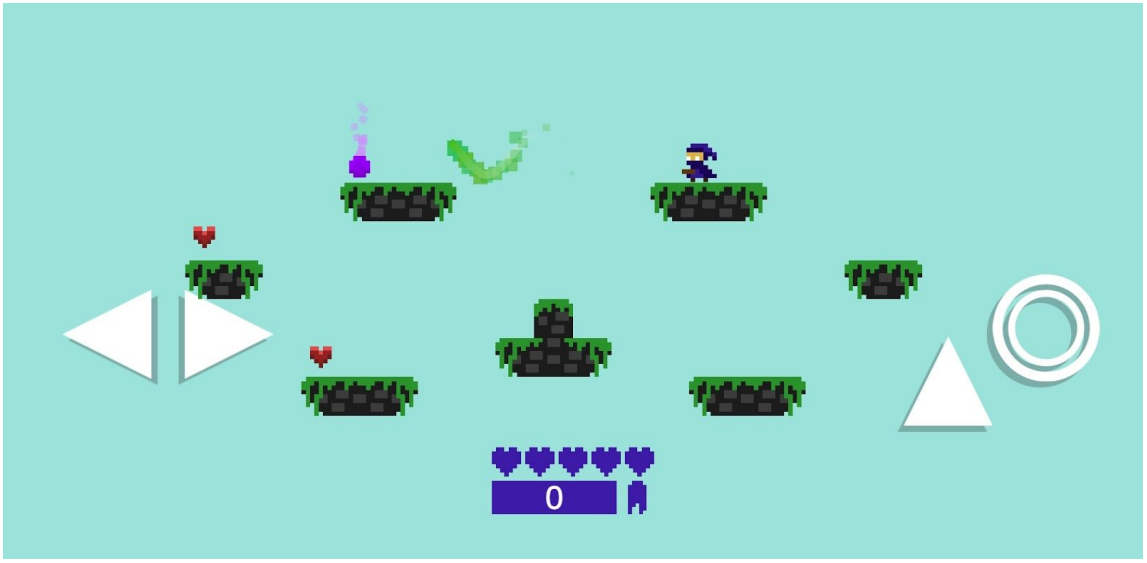
Relevant Scripts: *GameManager, PlayerManager.*

Mobile Controls

This project also features mobile controls. Since it's made in Unity, is 2D and has no major features tied to a specific platform, switching over between desktop and mobile is quite easy and fast.

The only difference in the game is in the **Game** scene, where there are two sets of on-screen controls.

- **Move** the player left and right.
- **Jump** the player in the air.
- **Attack** with the player's current spell.

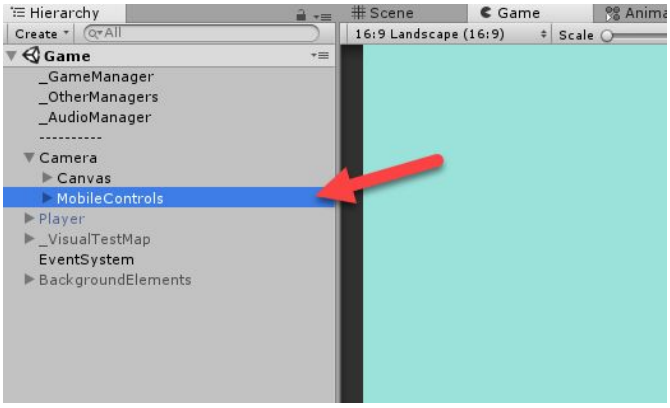


Each mobile control button is managed by a **MobileButton.cs** component. This checks for button down and up events, then triggers the corresponding input events.

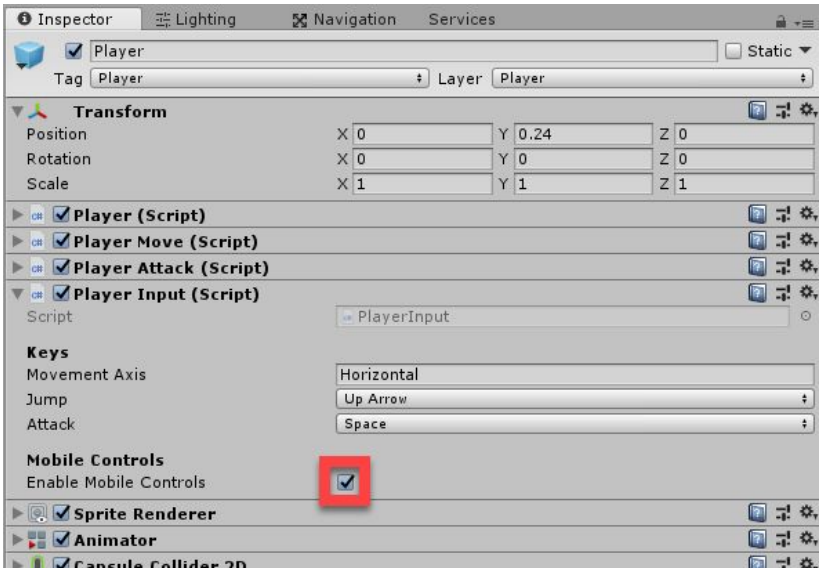
How to Enable Mobile Controls

By default, the project has mobile controls turned off. In order to turn them on, follow these steps:

1. In the **Game** scene, activate the **MobileControls** canvas (child of the Camera). You should see the mobile controls on-screen.



2. In the *Resources* folder, select the **Player** prefab. Then in the Inspector, enable **Enable Mobile Controls** in the **Player Input** component.



Relevant Scripts: *MobileButton.*

Folder Structure

- ❏ Animations
  - ❏ Menu
  - ❏ Player
    - ❏ PlayerBlue
    - ❏ PlayerGreen
    - ❏ PlayerPurple
    - ❏ PlayerRed
- ❏ Audio
  - ❏ Hits
  - ❏ Player
  - ❏ Spells
  - ❏ UI
- ❏ Fonts
- ❏ Photon Unity Networking
  - ➢ *Managed by Photon.*
- ❏ Physics Materials
- ❏ Plugins
  - ➢ *Managed by Photon.*
- ❏ Prefabs
  - ❏ Map Editor
  - ❏ Menu
  - ❏ Particles
    - ❏ Spell Hit Particles
  - ❏ Tiles
- ❏ Resources
  - ❏ Maps -
    - ➢ *Where maps are stores (don't change folder name).*
  - ❏ Pickups
  - ❏ Spells
- ❏ Scenes
- ❏ Scripts
  - ❏ Data
  - ❏ Managers
  - ❏ Map Editor
  - ❏ Menu
  - ❏ Misc
  - ❏ Network
  - ❏ Pickup
  - ❏ Player
  - ❏ Spell
  - ❏ UI
- ❏ Sprites

- ❏ Backgrounds
- ❏ Map Editor
- ❏ Particles
- ❏ Photoshop - *raw .PSD files*
- ❏ Player
- ❏ Spells
- ❏ Tiles
- ❏ UI
    - ❏ Mobile Controls
    - ❏ Spell Icons

Scripts

GameManager.cs
Manages the game during play. Checking for a win condition, etc.

NetworkManager.cs
Manages all the overall network stuff. Connecting to Photon, changing scenes, creating and joining a room, etc.

PlayerManager.cs
Manages all the players in the game. Finding a player, initializing them, etc.

MapLoader.cs
Loads in the map and contains info about the map like surface positions.

AudioManager.cs
Manages all the audio in the game - playing audio.

ParticleManager.cs
Manages all the particles in the game - spawning them.

SpellManager.cs
Manages all the spells in the game. Getting a spell, random spell, etc.

MenuUI.cs
Manages the overall menu. Changing screens, etc.

ConnectScreen.cs
Manages the connect screen - entering in a room code.

LobbyScreen.cs
Manages the lobby screen - players, buttons, etc.

SettingsScreen.cs
Manages the settings screen - changing name and volume.

GameSettingsScreen.cs
Manages the game settings screen - selecting a gamemode and map.

MapData.cs
Holds info about a map - tiles, spawns, name, etc. Saved as a JSON file.

MapEditor.cs
Manages the map editor. Selecting tiles, saving and loading maps.

MapEditorCamera.cs
Camera controller for the map editor. Move, zoom, pan, focus, etc.

BackgroundObjectMover.cs
Moves background elements like the clouds.

CameraController.cs
Moves the camera to track the players.

Shake.cs
Shakes an object. Used mainly for shaking the camera.

Serializer.cs
Converts a class to a byte array and vice versa. Used for transmitting data over network.

Pickup.cs
Checks for triggers with player and gives them the pickup. Attached to pickup object.

NetworkPlayer.cs
Top level class for each player. Contains ID, PhotonPlayer and player class.

Player.cs
Core class for the player. Takes care of health, damage, death, spawning, connects all other player components.

PlayerAttack.cs
Manages the player's casting of spells.

PlayerMove.cs
Manages the player's moving and jumping.

PlayerInput.cs
Manages local player input.

Spell.cs
Base class for a spell. Contains the *SpellData* class, caster, etc.

SpellProjectile.cs
Attached to a spell projectile object. Manages collisions and hitting players.

SpellProjectileData.cs
Data sent to the spell projectile once it's spawned.

SpellSelf.cs
Created when the player casts a self spell. Initiates the spell properties.

SpellContainer.cs
Player's container which contains a spell and the cooldown.

SpellData.cs
Data for a spell. Name, sfx, damage, speed, projectile, etc. All the info for a spell.

SpellOnHit.cs
Data class for on hit information

SpellOnSelfCast.cs
Data class for on self cast information.

SpellEditor.cs
Custom inspector for the *SpellData* class - so it's easier and simpler to use in the editor.

SpellHitData.cs
Data class that's sent to a player when they're hit by a spell.

GameUI.cs
Manages the countdown, header and win on-screen text.

PlayerUI.cs
Manages a player's UI hearts, score and spell.

UIButton.cs
Attached to all buttons. Hover scale change and sound effect.

MobileButton.cs
Attached to each mobile control button. Triggers the respective input event.

FAQ

If you have any questions you wish to ask about the project, feel free to contact me.

Error: The appId this client sent is unknown on the server
You need to create an account with Photon, then create an app in order to use their service. It's free, but there are paid options if you wish to expand. Learn how here.

How do I change the Photon server region?
Go to the PhotonServerSettings asset (*Window > Photon Unity Networking > Highlight Server Settings*) and change the *Region* to whatever region you want.

How do I build this game for a mobile device?
If you're new to building mobile games in Unity, you can learn from these resources, as it's quite a long answer:

- Android
- iOS

Glossary

Host
The player who created the game. Some processes and checks will go through them. They initialise the game and spawn the players.

Locally
Something happening *locally* means that it's only happening for you and not other players.

Globally
Something happening *globally* means that it's happening for all players in the game.

Client Side
If something is happening *client side*, that means it is being triggered / initiated by the client that causes it. E.g. if a player shoots a spell on their screen and it hits an enemy - it will register as a hit.

Server Side
If something is happening *server side*, that means it is being triggered / initiated by the master client. E.g. if a player shoots a spell on their screen, it only registers if it hits an enemy on the master client's screen.

RPC
Stands for *Remote Procedure Call*. These are functions that you can set to call on all clients computers in the game. Read more

Client
A player in a multiplayer game.

Master Client
A.k.a the *Host* - the master client is the player who created the game with some processes and checks going through them. They initialise the game and spawn the players.

Room
In Photon, a room is essentially a game / match / lobby. It's a self contained multiplayer connection between a group of players.

Photon
Photon is the networking framework used to create this game. Read more

Contact

Thanks for downloading the project! I hope you can find what you want in it. That being a project to learn Unity and multiplayer from, or as a template to kickstart your project. If you have any questions about the asset or find any bugs, feel free to contact me:

Email buckleydaniel101@gmail.com
Twitter @dbuckleydev

You can view my other assets I have up on the store here, and if you liked the project - reviews are much appreciated!

Changelog

v1.0

- Created the version 1.0 project.