Manual

What is EZFPS?

- EZFPS is a multiplayer first person shooter template for Unity3d, using Photon Unity Networking – a cloud based networking solution. EZFPS is designed to make a simple FPS Deathmatch style game, with Ai bots, zombies, player classes, weapon loadouts, and attachments. It is not specifically designed to be a Survival Game, as many have asked, however, there are weapon pickups that the player can collect in-game. As the player gets kills, they receive XP points, which then translate into a ranking. The ranking is displayed in front of the player's username in game. Using this XP and rank, certain weapons can be set to be unlocked as the player gains more experience. This creates incentive for users to keep playing, and adds diversity to your game. Rounds are free-for-all; as of now, no Team Deathmatch or other game-modes are available. Rounds are ended once one of the players reaches a set Score Limit (50 kills by default), and the leading player is given an XP Bonus. Headshots give an XP Bonus, as well.

How Do I Setup EZFPS?

- Tutorials are located at Assets/Aiden Studios/TUTORIALS
- Begin by following *importingEZFPS.pdf* to set up the Demo Scenes and Tags
- The rest of the tutorials, including *Adding Weapons, Changing the Player Model, Changing the Bot and Zombie Models, Adding a New Map, Adding a new Scope Attachment*, and more can be followed after.

Where Can I Find Help?

- I highly recommend reading through this documentation and the included tutorials, they, most likely, will answer your questions!
- The most preferable way of direct contact would be via the Unity Forums or Youtube . Posting a thread in the Forum will probably yield a faster response from me than a PM, and will give the community a chance to help you out, as well!
- If, for some reason, you cannot access one of those two, you can always email me at aidenclore@gmail.com

FAQ

- Q: Why aren't my weapon sounds playing?
  - A: Make sure you assign both the shooting and reloading sounds in the Tp Effect component, not just the Gun Script!
- Q: Why can't my game get past the Connecting/Loading screen?
  - A: Make sure Auto-Join Lobby is enabled in your PhotonServerSettings.
- Q: Why can Players see each others' cameras, or control each other?
  - A: On your Player Class Prefabs, make sure the CharacterControls component and the Camera game object are disabled by default.
- Q: Why are my bots clipping through objects or not moving at all?
  - A: Make sure to properly bake each Scene's Navigation Mesh in the Window > Navigation tab.
- Q: Why are my weapons not showing up, even though I Equipped them in the Loadout screen?
  - A: Make sure the Prefab Name on your weapon's wepItem matches the weapon's game object in the wepManager.
- Q: Why is my attachment not showing up after being equipped?
  - A: Same as previous question. Make sure your attachment is named the exact same thing as the model assigned in the wepItem's attachment arrays.

How does EZFPS Work?

- EZFPS is based on two main objects: your **_Room** and **Player Class Prefabs**.
  - **_Room:** There is one _Room game object in each scene, and it contains the *Room Manager* component. The Room Manager script is responsible for connecting to the network, displaying the Main Menu, Loadout Menu, Settings Menu, Server Browser, and Spawn Menu, in addition to keeping Score (both in-game score and XP), and preforming the Spawning process. The _Room object is where you will assign your spawn points, and scene-specific room information, like Score Limit, Class Prefabs. It is also what contains your universal Main Menu elements, like GUI Skin and Textures, Rank Items, Game Title, and Game Version. Each build can only play with its assigned Game Version variable, so old versions will not be compatible with newer versions.

roomManager.cs

Methods to Remember:

- JoinRoom(string roomName)
  - Joins the given room. In EZFPS, a Room's map is saved in the Room's Name. When you create a room, its name is set in this format: "Set Room Name | MapName"
  - The joinRoom() method splits the name it receives at the " |" character in the string. It then loads the scene based on the MapName, and joins the room. Without splitting the string and loading the new scene, your player would still join the room, but be in a completely different environment than the other player[s]. Coordinates will still update, so some players will appear to be flying or clipping through walls.
- quickPlay()
  - When called, the quickPlay() function will put the player in a room as quickly as possible. First, it scans if any rooms are available. If rooms are available, it chooses one randomly, and then uses JoinRoom() to join the room. If no rooms are found in the Room List, then it will create a new room with a random name, map, and bot and zombie count. The bot is chosen to be botAi[0] and the zombie is botAi[1].
- endGame(PhotonPlayer winner)
  - If you need to end a round prematurely, this is the method to call. The *winner* variable (*actually named "winn" in the class*) is the winning player.
- Spawn()
  - Spawns player at random spawn point
- SpawnBot(string botName)
  - Spawns the bot prefab named the given string. This is so you can add multiple bots to the botAi array, and spawn a specific one On-Demand.

Player Class Prefabs

- The *Player Class Prefabs* are your Player Prefabs for each of your playable classes. In order to work these prefabs (along with bot prefabs and explosions) **have to be in the Resources Folder**. Moving them out not allow them to spawn. By default, your Player Class Prefabs are located in *Assets/Aiden Studios/Resources*, and are named *Infantry*, *Scout*, and *Engineer*. When Instantiated, they are taken out of the Resources folder by name. That means that no two prefabs in the folder can be named the same. The main components are the *Character Controls*, *Move Anim,* and *Tp Effect* scripts. In addition, these prefabs need to have a *Photon View, Rigidbody, Photon Rigidbody View,* and *Audio Source*. The prefabs also need to have a game object with a Capsule Collider (body collision), another with a Sphere Collider (head collision), and a *recoilParent* in their children. The recoilParent contains the Camera, and the Camera contains the wepManager with your first person weapons.

Player Class Prefabs

- On the prefab, your CharacterControls component and Camera game object have to be **disabled**. Otherwise, players will be able to control each other and see each other's Camera view over the network. While the Camera has to be disabled, make sure your recoilParent game object is enabled. When the roomManager spawns your player, it locally enables the CharacterControls script and activates the Camera. For other players, the CharacterControls and Camera will still remain turned off, so they can still see your player's model, but cannot control it or see it's Camera view. The PhotonView syncs your player's rotation, and the Rigidbody PhotonView activates the Rigidbody on every player's computers, allowing for smooth movement (relative to just syncing the Player's position). The CharacterControls script controls your character, receives damage, controls dying, and contains the in-game Pause Menu. The moveAnim script plays the third-person animations that are shown over the network, and the TpEffect script syncs gun sounds, muzzle flashes, weapon models, and third-person reload animations. While both moveAnim and TpEffect contain variables for reloading animations, the moveAnim script just uses that as reference to pause the moving and idle animations when the reload animations are playing; the tpEffect script actually receives the command to reload and plays the animations. The moveAnim script uses the Rigidbody's velocity (which is synced over the network) as reference to play walking, sprinting, and idle animations. It uses the Body collider's height (also synced over the network) as reference for playing crouching animations. Using Rigidbody velocity and Collider height allows for each player's computers to locally play the models' animations, as opposed to one player's computer sending a Remote Procedure Call over the network every time it moves. This allows for better optimization.

CharacterControls.cs

Methods to Remember:

- [PunRPC] ApplyDamage(int dmg, string attacker, string coliderHit)
  - ApplyDamage() is a Photon Unity Networking Remote Procedure Call [PunRPC] that damages your player and syncs your health over the network. It also determines whether or not the collider hit was the head or not, and applies the head shot multiplier appropriately. If we call ApplyDamage() locally, the health would only go down locally. In order to have the health go down for everyone, we have to use a PunRPC. PunRPC's are called by using *PhotonView.RPC(method name, Photon Targets, parameters)*. So, as an example, if you want to damage yourself, you could say:

```
GetComponent<PhotonView>().RPC("ApplyDamage", PhotonTargets.AllBuffered, 250, "Nobody", "body");
```

- [PunRPC] Die()
  - This function, like ApplyDamage, is a PunRPC. Die() is called, by default, when the player's health is less than 1, but it can be called whenever you want a player to die. The method first spawns the ragdoll, then tells _Room that you've died and enables the GUI, sends the killer and victim's name to the kill feed, the attacker then receives a kill (via the *gotKill(string victim)* PunRPC, spawns the weapon drop object, and then finally destroys the user's Player Class game object.

tpEffect.cs

Methods to Remember:

- [PunRPC] shootEff(string audioClip)
  - Plays shooting effects over network. Network Instantiates muzzle flash for the third person model, and then loops through the gunSFX audio clip array to find which clip has the same name as the parameter, and then plays that clip.
- [PunRPC] updateModel(string wepName)
  - Loops through the third person weapon array (tpWeps) until it finds the object with the same name as the wepName parameter, and enables it. Tp Weps with names that don't match up are disabled.
- [PunRPC] reload(string audioClip)
  - First, loops through the gunSFX array and plays the audioclip whose name matches with the given parameter. Then, it checks if the player is moving or idle with the rigidbody.velocity variable, and plays the idleReload or walkingReload animation appropriately.

Shooting Mechanics

- The Camera located under the recoilParent in your Player Class Prefab contains all of the first person elements in EZFPS. The Camera's first child is the wepManager. This game object contains the Move Anim and Wep Manager components. The moveAnim script is the same script used for the third person animations. It reads the Player game object's Rigidbody velocity and plays the idle, walking, and sprinting animations appropriately. You'll notice that the script also contains "Reload", "Walking Reload", "Crouch", and "Crouch Walk" animation variables. **Do not** assign these variables, leave them empty. Any animations put in will probably not work or look very weird if assigned. They are designed for the third person model only. The animations you set change the wepManager's transform, which then moves all of the weapons below it. This allows for the weapons' shooting and reloading animations to play with the walking and sprinting animations seamlessly and simultaneously- removing the need for a walking-shooting or walking-reloading animation. The actual weapons contain several key components: the Gun Script (or Projectile Launcher), Aim Script, and Attach Man. The gunScript (or projectileLauncher) contains the bulk of all weapon functionality. including shooting, reloading, recoil, muzzle flashes, sounds, etc… The aimScript, obviously, controls aiming. It smoothly moves the weapons' transforms to their assigned Aimed Position when the the right button is clicked, and then back to the Default Position when it's released. The Attach Man is the attachment manager, and contains your Barrel and Sight attachments, which are activated and deactivated according the saved PlayerPrefs string, when the weapon is armed. The attachments themselves contain either a Scope Script, Silencer Script, or Light component. The Scope Script replicates the Aim Script, and overrides the weapon's Aim Script variables when it's active. The Silencer Script changes the Gun Script's shoot SFX and muzzle flash.

wepManager.cs

Methods to Remember:

- choosePrim()
  - Selects primary weapon. First, checks to see if the primary is reloading. If it is, it cancels the reload and resets the ammo back to its original count. Then, it sets the primary to active, and deactivates secondary.
- chooseSec()
  - Selects secondary weapon. Follows the same ammo procedure, but with the secondary. Then, activates the secondary and deactivates the primary.
- pickup(string wepName)
  - Reassigns the primary weapon to the wepName parameter. Loops through the primaries array and locates the weapon with the same name as the wepName, and then assigns it to the primary slot. If the weapon being assigned is already the primary, it adds 3 clips to the magazine count.
- meleeAttack()
  - Does melee attack. First, checks if the knife animation and swipe animation clip are assigned. If they are, it makes sure the knifeSwipe animation isn't playing. It then plays the knifeSwipe animation and recoil, if the recoilScript is assigned, and tells the tpEffect script to play the knife sound through the "shootEff" PunRPC. Like the zombieScript, it then shoots a raycast from the player's origin forward with a short range, applies damage, and spawns particles appropriately.

gunScript.cs

Methods to Remember:

- Fire()/FireSingle()
  - The shooting function. First, it makes sure you aren't on cool down, that you aren't reloading, and you aren't drawing your weapon. Then, it checks if you're ammo is greater than zero. If it isn't, it calls the reload function. If it is, it plays the shooting animation, subtracts one from the ammo, sets the cool down, plays the recoil, sends the shoot method to the tpEffect script, spawns the muzzle flash, and then shoots a raycast. If the "pelletSpawns" transform array is not empty, it will shoot a raycast from each one of the assigned transforms. If the array is empty, it ill just shoot a raycast from the "forward" transform with the assigned range. If the raycast hits an object with the "Player" tag and has a PhotonView component, it will send an "ApplyDamage" RPC to that PhotonView, with the a random variable between the max and minimum damage variables, your name, and the name of the collider it hit. It then spawns the blood particle prefab. If the object isn't tagged "Player", the dirt particle prefab is instantiated.
- Rel()
  - The reload function. If the current ammo is not maxed out, and you aren't out of magazines, this method sets the ammo to the clip size, plays the reloading animation, takes away a magazine, and tells the tpEffect object to "reload", with the reload sound.

projectileLauncher.cs

Methods to Remember:

- Fire()/FireSingle()
  - The shooting function. First, it makes sure your shooting, reload, and draw animations aren't playing. Then, it makes sure you have ammo. If you don't, it calls the rel() function. If you do have ammo, it plays the shoot animation, subtracts one from your ammo, spawns your muzzle flash, instantiates your projectile, and then adds forward force to it, launching it at the assigned shoot speed. Finally, it tells the tpEffect object to play the "shootEff", with the gun sound, and sends the recoil information to the camera.
- Rel()
  - The reload function. If the current ammo is not maxed out, and you aren't out of magazines, this method sets the ammo to the clip size, plays the reloading animation, takes away a magazine, and tells the tpEffect object to "reload", with the reload sound.

camRecoil.cs

Methods to Remember:

- StartRecoil(float recoilParam, float maxRecoil_x, float speed)
  - Sets local variables to the parameters.
- Recoiling()
  - Checks if the recoil float is greater than 0. If it is, locally change the rotation of the transform according to the given parameters, and set the recoil float to subtract time. If recoil is not greater than 0, set it to 0 and lerp the object's rotation back to its resting position. This method is called in the Update() function.

wepPickup.cs

- This script is to be placed on a weapon model with a Box trigger collider attached to it. When a player enters the trigger, it activates a Boolean inTrigger. If inTrigger is true, and the PhotonView belongs to the local player, a GUI message appears saying “Pickup Weapon Name” or “Buy Weapon Name”, depending on what you set the weapon’s price to be. If the player presses ‘E’ while in trigger, it calls the “pickup” function on the player’s Wep Manager component, with the selected wepInfo’s prefab name. If the bool “destroyOnGet” is true, it will then destroy the game object via PunRPC over the network. If you want the weapon to just be a pickup, you can add a Rigidbody component, solid collider, price equal to 0, and destroyOnGet to true. If you want it to be a wall purchase, like COD Zombies, just have the gun float on a wall, set the price higher than 0, and set destroyOnGet to false.

Bot Mechanics

- Bot game objects contain three main components: Dummy Ai, Bot or Zombie Ai, and Tp Effect. In addition, bots require a Photon View, Rigidbody, Animation, Audio Source, Capsule Collider, and Nav Mesh Agent. It's important to note that, since the bots use the Nav Mesh system, you need to bake your scenes' Navigation Meshes. The dummyAi script is what handles your player's damage and dying functions. The ApplyDamage() and Die() functions are the same as the CharacterControl's, and they're referenced in the same way. The botAi script is used for ranged bots, and controls setting the navigation path, targeting, shooting, reloading, and other bot functions. The ZombieAi script is extremely similar to the BotAi one, but has a melee attack, no reload, and will not target other zombies. In addition, Zombie's have a "find radius", where when a player comes within a certain radius, the zombie will target them. The targets of both scripts are assigned to the closest player, bot, or zombie. Bots will attack each other and players, unless zombies are on the map. If zombies are on the map, bots will only target players or other bots if they are damaged by them. Both Bots and Zombies shoot just like the gunScript. They fire a raycast from their origin facing forward, and if the raycast hits something tagged "Player" with a PhotonView, it sends an "ApplyDamage" PunRPC. Zombie's obviously have a much lower range, however. The Rate of Fire is determined by the shoot animation. The faster the shoot animation, the faster the rate of fire. The potential targets are any game object that contains a Tp Effect component, for bots and players, and any object with a Zombie Ai component for zombies.

dummyAi.cs

Methods to Remember:

- [PunRPC] ApplyDamage(int dmg, string attacker, string collider)
  - Same exact thing as the CharacterControl's ApplyDamage(). Determines whether or not the hit collider was the head, and then subtracts health accordingly.
- Die()
  - Spawns Ragdoll, sends "gotKill" to killer, and then destroys the game object through a second PunRPC called "killMe". By default, called when health is less than or equal to 0.

botAi.cs

Methods to Remember:

- [PunRPC] SetTarget(string name)
  - Checks if a game object with the given name string exists. If it doesn't, calls findNearest(). Otherwise, if it does exist, sets the target variable as that game object's transform. The target variable's position is set as the Nav Mesh Agent's Destination in the Update() function.
- findNearest()
  - First, sees if zombies are in the room. If there are, loop through the zombies array (gameobjects with zombieAi script), and use Vector3.Distance to find the closest game object in the array. Then sends PunRPC "SetTarget" with the that name. If no zombies are present, loop through all objects with a tpEffect instead, and set the target appropriately.
- fire()
  - Very similar to the gunScript's fire() function. First, makes sure that the shooting animation isn't playing, the target still exists, and the object is at the Nav Mesh's stopping distance. Then, it makes sure that the reload animation isn't playing , and the ammo is greater than 0. If there's no ammo, call the reload function. If there is, subtract 1 from the ammo, and fire a raycast from the transform's origin pointing forward with the selected range. If the ray hits an object with a "Player" tag and that contains a PhotonView, send an "ApplyDamage" PunRPC with a random integer between the minimum and maximum damage, the bot's name, and the hit collider. Also, play the shoot animation and tell the tpEffect to shoot.
- reloadM()
  - Plays reload animation, sets ammo to clip size, and plays reload SFX.

zombieAi.cs

Methods to Remember:

- [PunRPC] SetTarget(string name)
  - Same as botAi.
- findNearest()
  - Loops through potential targets, compares Vector3.Distance(), and sets the closest object as the target through setTarget(). Potential targets are set to any game object with a Tp Effect component.
- fire()
  - Same as botAi's. By default, the range is set to be much lower, to make it a melee attack, however.

Menu Items

- Menu Items are prefabs that are never actually spawned in-game. They are simply a means of holding information. The current menu items are
  - mapItem
  - wepItem
  - rankItem
- Each component is designed to be placed on an otherwise empty game object and turned into a prefab. The roomManager, and other scripts, read these objects and pick out the information they need. For example, in the roomManager, the script looks at every mapItem set in the maps array, and takes the Icon, Score Limit, and Size variables to display the GUI, and then uses the Level to Load variable to load the actual map from the Scene Manager. This theme is repeated throughout the script, and rest of the project. The wepItems also contain arrays for attachments, which are used in the GUI and setting the PlayerPrefs string to activate attachments accordingly. Because we are using PlayerPref strings, the objects used, like attachments and weapons, have to be named exactly the same between menu and game. That's why, in the wepItem script, there's a Prefab Name and Display Name. The Prefab Name is going to be the name that's actually used to be looped through the primaries in the wepManager, whereas the Display Name is what's going to be shown on the GUI. The Level to Load string in the mapItems has to be the same name as the Scene you want to load too, for this same reason. In the rankItems, the XP variables are compared against each other to see which lines up with the player's XP PlayerPref. For the one that does line up, that rankItem's icon and rank name are drawn in the GUI, and the abbreviation is added to your player name when you join rooms.

killFeedChat.cs

Methods to Remember:

- [PunRPC] Chat(string message, PhotonMessageInfo messageInfo)
  - Adds the message string to the chat feed, and syncs across clients.
- [PunRPC] AddFeed(string victim, string killer)
  - Adds a message following the layout: killer [FRAG] victim
  - Called when player gets kill.
- [PunRPC] promo(string name, string rank)
  - Adds a message with the layout: name was Promoted to rank!
  - Called when player is promoted

Other Components Rundown (1/4)

- moveAnim.cs
  - Uses Rigidbody Velocity to play idle, walk, and sprint animations for first and thirdperson models.
  - Uses Body Capsule (assigned in the Character Controls component fetched from the rb (rigidbody)'s gameobject's height to appropriately play crouching animations in **third person** models. (Do not assign for first person i.e wepManager object)
- mouseLook.cs
  - Typical Smooth Mouse Look script. Uses Mouse's X and Y axis to rotate camera and assigned Character Body game object appropriately. If set through the menu, the sensitivity is set to the "mouse" PlayerPrefs float. If PlayerPrefs doesn't contain "mouse" yet, it defaults to the values assigned in the Inspector.
- headBobber.cs
  - Bobs object up and down based on horizontal and vertical Input Axes. Uses simple transform translation to move object based on Bobbing Amount and Speed, midpoint, and Sprint Multiplier.
- ragDoll.cs
  - Ragdoll manager, to be placed on both Player and Bot ragdoll prefabs. Player Ragdolls require an fpCam (First Person Camera) variable, and bot ragdolls must not have one. If the fpCam exists, this script destroys it after 3 seconds. The CharacterControls script enables the fpCam locally, so the game object must be disabled by default on the prefab. The script sends a PunRPC to itself to destroy the game object after 15 seconds. In addition, if the "grunts" Audio Clip array has any clips assigned, the script will play a random one when spawned.

Other Components Rundown (2/4)

- aimScript.cs
  - When the Right Mouse Button is held down, translates the object's position the Aimed Pos Vector3's coordinates, and zooms in the Camera's FOV to the Aimed FOV. When released, moves the object back to its hip, and the Camera's FOV lerps back to default. In addition, sensitivity is changed when the object is zoomed in, and the Scope Texture will be drawn, if assigned, when the Camera's FOV reaches the Aimed FOV. The Smooth Aim variable adjusts the speed. When the Scope Texture is drawn, each object in the Objects to Hide array will be disabled, and renabled when un-aimed.
- attachMan.cs
  - Finds and enables the correct attachments based on the proper PlayerPrefs string (weaponName + "sight" or weaponName + "barrel"). If the sights PlayerPref string isn't set, the Default Sight object will be enabled. The method setAttachment() loops through the sight and barrel Attachments arrays to find the one with the same name as the PlayerPref, and enables it. All other objects in the arrays are disabled.
- explosionDmg.cs
  - Applies explosion damage based on radius. Uses Physics.OverlapSphere to and Vector3.Distance (in the script is actually just (location – col.transform.position) to find all colliders in radius, determine if they are tagged "Player" and contain a PhotonView, and then sends an "ApplyDamage" PunRPC with calculated damage based on proximity and the assigned dmg variable. Made to be placed on explosion particle game objects.

Other Components Rundown (3/4)

- projectile.cs
  - To be placed on projectile game objects. Contains two main PunRPC methods: moveFwd and explode. moveFwd is called by the projectileLauncher script, and adds force to the game object's rigidbody over the network. The explode() RPC is called, by default, OnCollisionEnter(), but can be changed to be set whenever. The explode method Instantiates the impact prefab over the network (designed to be an explosion), destroys itself, and sets the impact game object to be destroyed after three seconds.
- scopeScript.cs
  - Designed to be placed on Sight/Scope attachments. Script reassigns the variables of an assigned weapon's aimScript on awake. It overrides the Aim Pos, Aim FOV, Scope Texture (optional), and ObjectsToHide (optional), on the OnEnable() function.
- silencerScript.cs
  - Designed to be placed on Silencer attachments. Script reassigns a gunScript's shoot SFX and muzzle flash. Currently does not work with projectileLauncher script, but can be done easily by changing the "weapon" variable from a gunScript, to projectileLauncher type.
- wepSway.cs
  - Uses Mouse Input Axes to locally rotate weapon transform smoothly, replicating realistic weapon sway when the mouse is being moved. Designed to be placed on weapon's "sway" child, not main model with the gunScript. If placed on the main weapon object, it could interfere with the aimScript and glitch out. If placed on the weapon's model with the Animation Component, the animations will override the sway.

Other Components Rundown (4/4)

- smoothFollow.cs
  - Designed to be placed on the _Room object's Camera. The script adjusts the game object it's attached to's transform to follow a specified target, for spectating. By default, the roomManager chooses a random from its targets to spectate, but when the _Room object receives the Died() method, it sets this script to follow the assigned killer. This script receives the GUI Skin from the _Room object, and uses it to display the "Observing: target's name" GUI Box. The actual following portion is done by simple Transform repositioning and rotation adjustment, which is lerped for smooth movement.