**DESIGN DOC II**

**<ARMAMENT>**

****

**REVISION HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| Revision # | Author | Revision Date | Comments |
| 1.0 | Andrew Pitt | Feb. 18, 2019 | Initiated |
| 1.1 | Andrew Pitt, Jeremy Scott, Alex Cohn, Keith Bosworth, Ben Mankin | Feb. 28, 2019 | Added general requirements and class descriptions |
| 1.2 | Andrew Pitt, Jeremy Scott, Alex Cohn, Keith Bosworth, Ben Mankin | Mar. 1, 2019 | Minor fixes |
| 1.3 | Andrew Pitt, Jeremy Scott, Alex Cohn, Keith Bosworth, Ben Mankin | Mar. 1, 2019 | Edited and fixed formatting |
| 2.0 | Andrew Pitt | April 27, 2019 | Minor revisions and edits |
| 2.1 | Keith Bosworth | April 28, 2019 | Update system overview |
| 2.2 | Alex Cohn | April 28, 2019 | Additions/Revisions |
| 2.3 | Andrew Pitt | April 28, 2019 | Formatting and clarifications |
| 2.4 | Andrew Pitt | April 28, 2019 | Weapons and Items related additions |
| 2.5 | Ben Mankin | April 29, 2019 | Accepted Revisions |

**Table of Contents**

[**System Overview**](#_cf4hkiduq5v) **5**

[**System Block Diagram**](#_6jpincpip7sy) **9**

[**Glossary**](#_vh4snlalkewr) **9**

[Document Overview](#_43uthbua0pey) **10**

[Software Architecture](#_in9u8jem9q60) **10**

[CameraAnimationHandler Class](#_3r26dqxyx03l) 10

[AICharacterControl Class](#_xzdeekve474n) 11

[AIEnemyWatcher Class](#_mrizbuxbzgyx) 14

[AIGunWatcher Class](#_j3fkvath28qw) 15

[FragGrenade Class](#_x5637lkoveq1) 15

[FragExplosion Class](#_7g2zdrrhzqd1) 18

[Gun Class](#_y52wh4nqzkc1) 19

[Medkit Class](#_377m5d8bosnf) 20

[Chat Manager Class](#_m784xuoi71fj) 22

[CountdownTimer Class](#_cijsrv6apvxn) 24

[JumboTronDisplay Class](#_q63f54cp5gk7) 25

[MobileButtonController Class](#_604popueukn5) 26

[PlayerAnimatorManager Class](#_us2uzwmrhnrm) 26

[PlayerNetworkMover Class](#_gky8q1besl4v) 27

[WallDropAnimator Class](#_lzwbyxzuvgz) 28

[WallManager Class](#_4800onetjcxu) 29

[WallTarget Class](#_d4evgh409r8r) 30

[WeaponsMenuManager Class](#_sdyzd03m8n5m) 31

[EmailInputField Class](#_a4qmano3uire) 32

[SpinModels Class](#_tqexmk4plt79) 33

[PlayFabController Class](#_sh4rcxy35rr) 34

[public Text playerNameText: The associated player’s name .](#_jq3dwusqtjii) 49

[public Text playerScoreText: The associated player’s score, depending on the leaderboard presented.](#_9kww8j39foe5) 49

[Launcher Class](#_fystyghlf318) 49

[PlayerManager Class](#_ys64gh7sli7o) 52

[PlayerStats Class](#_ui1key9ktlal) 55

[PlayFabManager Class](#_7zejbwxmblgo) 55

[IShootable Interface](#_pkbhatzce0d8) 57

[IThrowable Interface](#_azhpb0om5swn) 58

[IUseable Interface](#_byt0bk7knn04) 58

## System Overview

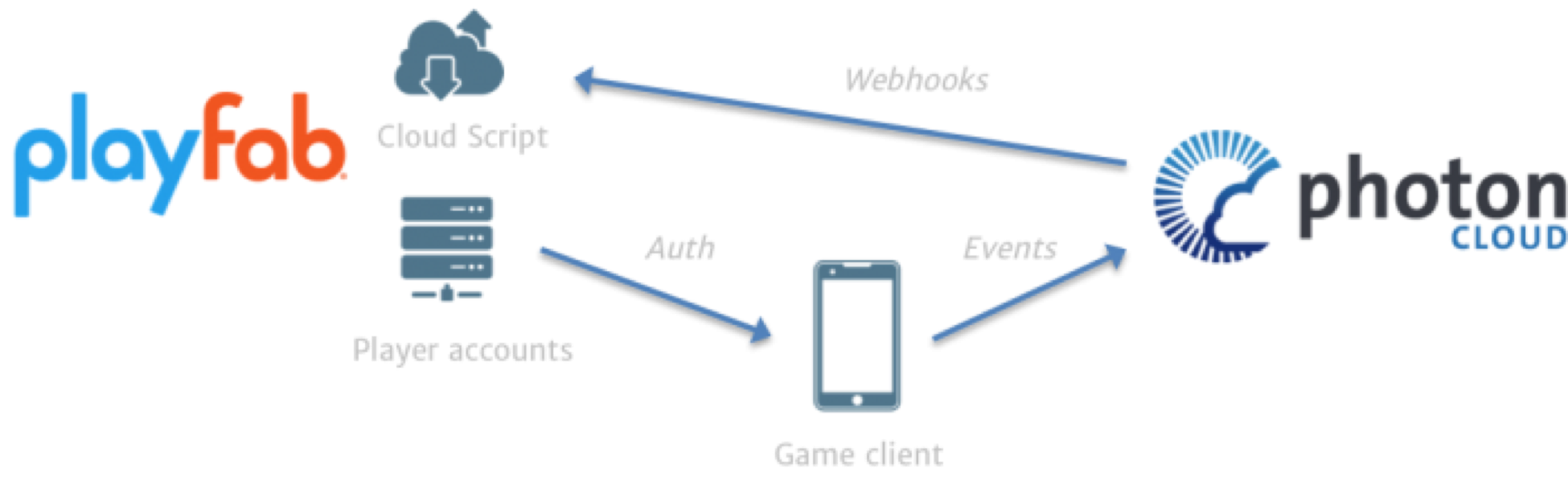
This section provides an overview of the Armament project: an original, networked, multiplayer, team-based video game built for PC, Mac, iOS, and Android platforms. Armament is built with the Unity game engine and API, Photon Networking servers and API, and PlayFab backend database and API.

Gameplay will consist of two stages played in succession: first, an *Armament* stage where players gather weapons and resources, and subsequently, a *Battle* stage where combatants fight for control of the arena using the resources they’ve acquired. Original sounds and art will be created for the project in addition to existing assets.

Armament will be designed and driven with the Unity engine and API, which contains numerous scripts and libraries that provide abstraction for the low-level details of physics rendering, graphics processing, animation, A.I, platform-specific builds, and system analytics.,

Player information is stored in a database provided by PlayFab. Users can register accounts and authenticate from both PC and mobile devices. Once logged in, players will see the Launcher where they can choose to play a game, or they can check the statistics stored in the database, which may be accessed through the leaderboard. If they choose to play, their statistics during that game will be updated to the leaderboard. Players can also add friends through the Launcher, which will allow them to invite those friends to private games. This feature gives players the ability to stay in touch with players they enjoyed playing with.

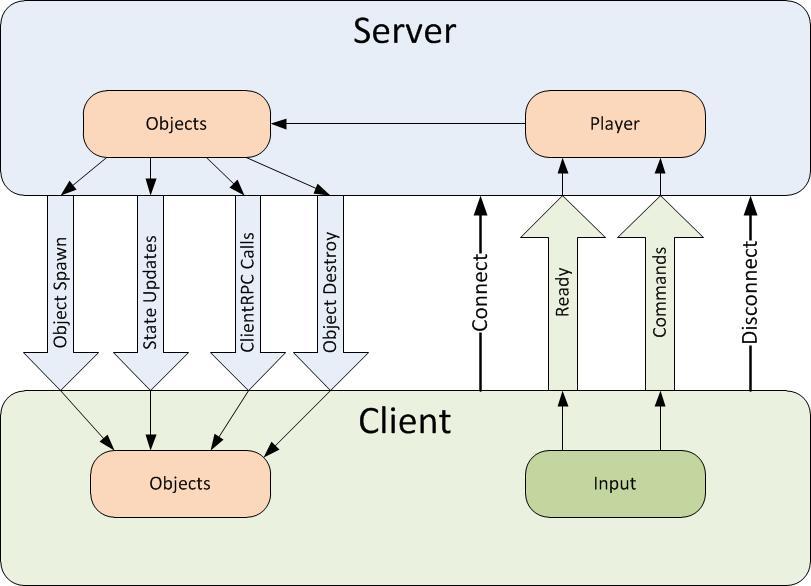
The interaction between Unity clients, the Photon cloud, and PlayFab can be seen below in the abstraction provided by **Figure 1.**

****

**Figure 1.** General overview of Unity client integration with Photon and PlayFab from a mobile perspective (similar for PC).

To play the role of the serving or host device, one client will assume the role of master client, which will synchronize remote clients by broadcasting RPC through Photon servers. A master client is a specialized type of local client that synchronizes remote clients using Photon servers as a means of communication, which sends and receives changes in specific game state based on client input. This implementation is headless in that it only passes RPC calls and choice data through Photon servers, as opposed to processing graphics and physics data within the Photon server space.

In order for Armament to synchronize across every connected device, copies of each object created in the game will exist in memory on both the master and remote clients. The master client device assumes the role of the authority to keep track of changes made to various GameObjects, as well as communicating those changes to the clients. The client/server relationship, RPC calls, data flow, GameObject storage, and state changes can be seen in **figure 2** below:



**Figure 2.** Overall data flow, object storage, and state changes from server (master client) to remote and local clients.

Data will be broadcasted from client to client using RCP and other modes of communication and data synchronization via the Photon server, and through the internet via UDP and TCP. The Photon networking framework provides a robust networking API built specifically for Unity projects in order to meet networking requirements. Clients send input, which is received by listening to various RPC events (for example OnMouseClick() called from within an RPC wrapper to listen to mouse clicks supplied from user input). The Photon Networking API provides abstraction of low-level socket code. **Figure 3** describes the user flow to begin playing online with other players.



**Figure 3.** User flow to begin playing Armament

When joining a game, players must use the in-game menu to connect to a Photon *name server*, which gives them access to a *master server*. Master servers are geographically located around the globe to provide low ping times to all clients, regardless of their location. Master clients will then place the clients in a *master server for matchmaking purposes.* . When a player finds a match they will communicate with other clients via a Game Server that is responsible for hosting the game room they are in.

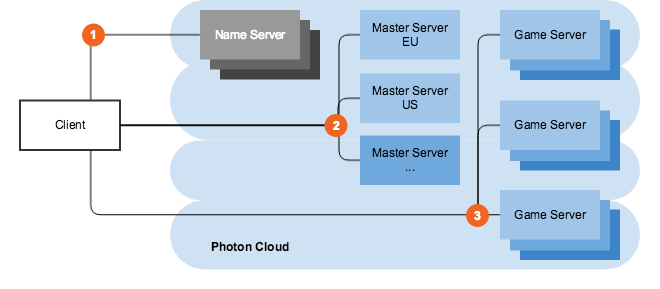
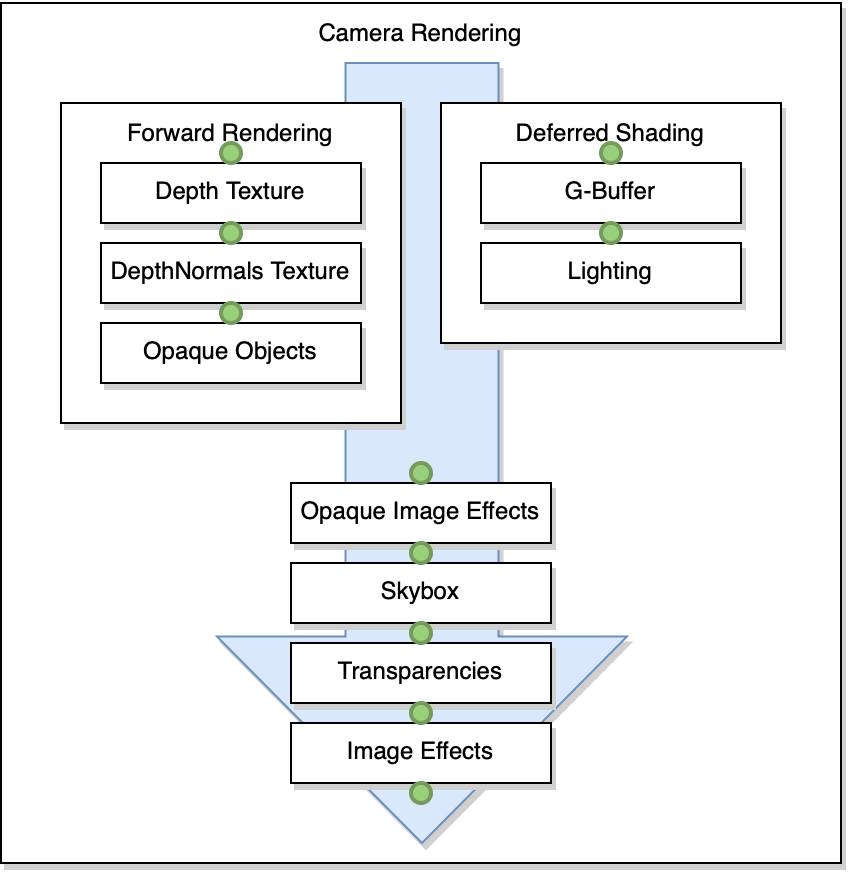


Figure 4.Master server and game server layout..

In the case that the host connection quality terminates or becomes suboptimal, the Photon API offers a host migration service which is called in order to move server identity to the next available device. Sending RPC through the Photon server adds one layer of security for clients in that the IPs of remote clients are managed by the Photon server instead of seen directly by the master.

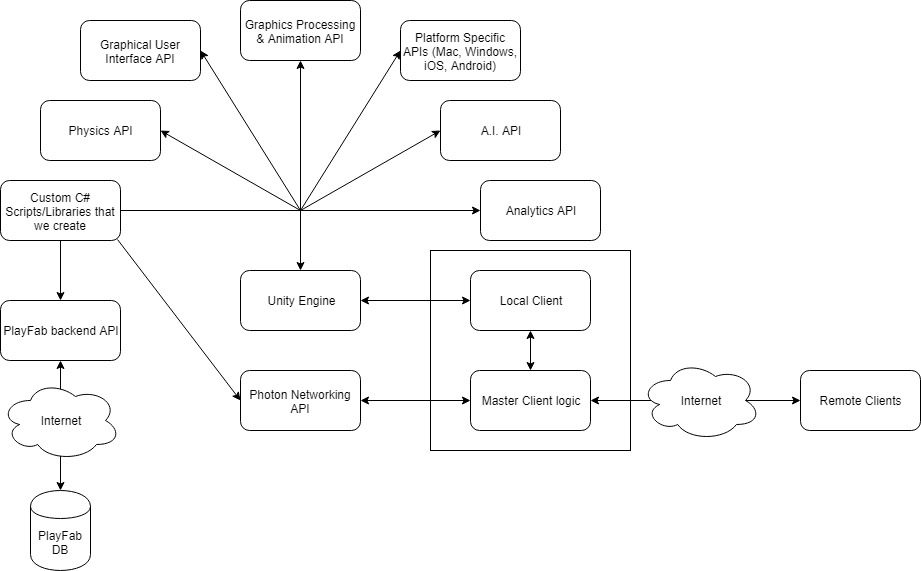
The onus of graphics processing is placed on the client devices, which may flex the master client’s processing capabilities somewhat, but overall is not an impediment considering how picky our master client is in terms of choosing what to broadcast to clients through Photon servers. On the client side, the Unity engine will render a camera placed in the virtual environment, resulting in the data flow represented in figure 5 below. After graphics processing, certain state changes to the GUI will be communicated as attributes via various synchronization mechanisms provided by photon to all clients over the Internet.



**Figure 5**. Camera rendering data flow

During gameplay, players have the option to toggle an A.I. controller. When this controller activated the game automatically takes control over the player avatar’s movement and actions. If the A.I. controller is activated during the *Armament* stage, the A.I. player will target (i.e., go to) known gun spawn points in order to pick up guns. Along the way, the A.I. may recognize that a gun, which it is not currently targeting for pickup and not yet picked up by another player, has come into view. . When it sees a valid gun target, it will run towards it and attempt to pick it up. During the Battle stage, the A.I. will wait for opponents (i.e., players on the other team) to come into view or shoot at it. Either event will trigger the A.I. player to target the opponent. The A.I. player will pursue its targeted opponent even if the player tries to run away. If the A.I. player gets the target in its crosshairs, it will immediately shoot..The A.I. player always calculates shortest path to its destination whether or not its target’s position changes. A high-level representation of the interlocking systems is described below in figure 4.

## System Block Diagram



**Figure 4.** A high-level view of the components in Armament.

## 

## Glossary

* **Master client**: the client that is designated to act as a pseudo server for all other clients. The master client becomes responsible for making decisions and coordinating actions that would typically be the responsibility of a server in a server-client model. Any client that joins a game room can potentially become the master client at some point. By default, the master client is chosen in the order of who entered the game room first.
* **Remote client**: all clients that are not currently the master client.
* Name server: the first server that every client contacts, which provides the list of available regions.
* **Master server**: every region has a completely separate master server for matchmaking.Game server: hosts game rooms
* **Launcher**: the first scene presented to the user upon starting the game. In this scene, a user has the ability to log in to their account, choose their gameplay options, and enter a game room to begin playing the game.
* **Spawn**: instantiation of a GameObject within the scene.
* **Prefab**: a “prefabricated” set of GameObjects that are linked in a parent-child relationship. Prefabs have components that affect the way they behave (as it appears to the user) after graphics processing.
* **Components**: scripts which can be added to GameObjects in order to change the GameObject behavior, appearance, and properties. The Unity API contains certain fundamental components that can be added to a project. Developers also have the capability to write original scripts as components.
* **Lobby**: appears to the user as a GUI panel with buttons that can connect them to other players in order to begin gameplay.
* **Room**: the gameplay environment.
* **Health**: a value that represents in-game measure of life. Players start with 100% health. When players reach 0 health, they have died and are removed from the game.
* **CoolDown**: The time it takes for an item to become available for use. Example: for a gun, the cooldown is the rate of fire.

## Document Overview

The software design document describes each C# class necessary to create in order to meet the requirements. Outlined below is the purpose of every class, method, and data field, along with the return values, exceptions, preconditions, and postconditions associated with each method.

## Software Architecture

### CameraAnimationHandler Class

**Class Purpose**

Manages the camera that follows a player and provides the user a first-person view within the environment.

**Data Fields**

**private Transform lookPoint:** Coordinates (x,y,z) of the camera position and the range of its view.

**private float rotateSpeed:** The rate of change at which the camera can rotate on any axis (x,y,z).

**Methods**

**public void Update():**

**Purpose:**

* Overridden from class Monobehavior (Unity API). Called each time the graphical frame is updated.

**Preconditions:**

* Start() has returned.

**Postconditions:**

* GUI is updated

### AICharacterControl Class

**Class Purpose**

Controls the actions of a player when under AI control.

**Data Fields**

**public NavMeshAgent Agent**: the navmesh agent required for the path finding

**public ThirdPersonCharacter Character**: the character we are controlling

**public Transform target**: target to navigate to

**PlayerManager pm**: keeps a reference of the player manager attached to the player GM

**Camera fpsCam**: keeps a reference to the camera on the player

**private bool isStage1**: flag to keep track of the stage in the game

**private int[,] graph**: the graph of distances used in dijkstras algorithm

**private int V**: keeps track of the number of vertices in dijkstras algorithm

**private int[] dist**: keeps track of the distances of the root to all the vertices in the graph

**Transform[] transforms:** locations to visite

**bool[] visited**: keeps track of what locations have been visited

**int currentTargetTransformIndex**: used to keep track of the index in the array of the location AI player is currently targetting

**string lastOutputType**: used for debugging

**Methods**

**public void Start():**

**Purpose:**

* Initializes member variables. Creates distance graph to all weapon spawn points. Finds the distances to all the guns.

**Preconditions:**

* Player has been instantiated and AI is toggled on.

**Postconditions:**

* Creates distance graph to all weapon spawn points. Finds the distances to all the guns.
* The script is now active.

**void CreateDistanceGraph():**

**Purpose:**

* Creates distance graph for weapons on player’s team’s side of the arena.

**Preconditions:**

* Player and weapons have been spawned. AI is toggled on.

**Postconditions:**

* Graph variable is populated with distances.

**void OnStage1TimerIsExpired()**

**Purpose:**

* Handles the event that stage 1 timer expires.

**Preconditions:**

* Stage 1 timer expired.

**Postconditions:**

* Member variables are reset.

**void OnStage2TimerIsExpired()**

**Purpose:**

* Handles the event that stage 2 timer expires.

**Preconditions:**

* Stage 2 timer expired.

**Postconditions:**

* Distance graph is recreated.

**public override void OnEnable()**

**Purpose:**

* Handles the event that this component has been enabled.

**Preconditions:**

* This component has been enabled.

**Postconditions:**

* Stage timer callbacks have been set.

**Transform FindNextPointToTaget()**

**Purpose:**

* Finds the next point to target.

**Preconditions:**

* Distance graph has been created.

**Returns:**

* The next transform to target.

**void OnTriggerEnter(Collider Other)**

**Purpose:**

* Handles the event that this play has collided with another collider with a trigger.

**Preconditions:**

* Player has collided with a trigger such as a weapon.

**Postconditions:**

* Player no longer targets that object.

**void AccomplishStageBasedGoals()**

**Purpose:**

* Accomplish stage based goals.

**Preconditions:**

* Called on every update.

**Postconditions:**

* Stage based goals are set**.**

**public void Update():**

**Purpose:**

* Callback for every frame update.

**Preconditions:**

* Start() has returned.
* New frame is being processed.

**Postconditions:**

* AccomplishStageBasedGoals has been called.

**private bool EnemyIsInCrosshairs(out PlayerManager enemy)**

**Purpose:**

* **Raycasts from player's camera to check if enemy (player on a different team) is in this player's crosshairs.**

**Preconditions:**

* None.

**Returns:**

* True if enemy is in crosshairs and false otherwise.

**private void ShootGun()**

**Purpose:**

* **Shoots active gun if player has an active gun to shoot.**

**Preconditions:**

* Player has active gun to shoot.

**Postconditions:**

* Player shoots gun on the network.

**public void SetTarget(Transform target)**

**Purpose:**

* **Sets target.**

**Preconditions:**

* None.

**Postconditions:**

* Target is set.

### AIEnemyWatcher Class

**Class Purpose**

Keeps track of what enemies are in view of the player.

**Methods**

**public void OnBecameVisible():**

**Purpose:**

* Callback that handles when this player becomes visible to a camera.

**Preconditions:**

* This player becomes visible to a camera.

**Postconditions:**

* Locally controlled player’s EnemiesInView list is appended.

**public void OnBecameInvisible():**

**Purpose:**

* Callback that handles when this player becomes invisible to a camera.

**Preconditions:**

* This player becomes invisible to a camera.

**Postconditions:**

* This player is removed from local player’s EnemiesInView list.

### AIGunWatcher Class

**Class Purpose**

Keeps track of what guns are in view of the player.

**Methods**

**public void OnBecameVisible():**

**Purpose:**

* Callback that handles when this gun becomes visible to a camera.

**Preconditions:**

* This gun becomes visible to a camera.

**Postconditions:**

* Locally controlled player’s UnclaimedGunsInView list is appended.

**public void OnBecameInvisible():**

**Purpose:**

* Callback that handles when this gun becomes invisible to a camera.

**Preconditions:**

* This gun becomes invisible to a camera.

**Postconditions:**

* This player is removed from local player’s UnclaimedGunsInView list.

### FragGrenade Class

**Class Purpose**

Represents a fragmentation grenade that can appear in the arena or in a player’s inventory when picked up from a location in the arena. Players may throw the grenade once picked up.

**Data Fields**

**public float timer:** The value which the timer will be set to every time.

**public float countdown:** This countdown represents the grenade explosion timer. Initially, it is set to the timer value, and then decremented to 0 over a series of frames.

**public float explosionRadius:** The radius in graphical units of the grenade explosion.

**public float explosionForce:** The force to be applied by the explosion.

**public float baseDamage:** The base damage caused by the grenade, to be scaled by distance ftrom the grenade.

**private float distanceFromGrenade:** distance of objects hit by grenade explosion.

**public float damageCaused:** Determined by base damage and distance from grenade.

**public bool thrown:** Boolean that indicates whether or not the player has thrown the grenade.

**public bool hasExploded:** Boolean that indicates whether or not the grenade has exploded.

**public bool grenadeWasPickedUp:** Boolean that indicates whether or not the grenade has been picked up.

**[SerializeField] GameObject explosionParticle:** The particle effect that represents the explosion.

**public MonoBehaviourPun playerWhoOwnsThisGrenade:** a reference to the player who owns this grenade.

**Methods**

**void Start():**

**Purpose:** called once and only once at the beginning of the script’s lifecycle. Used to set values before Update() begins.

**Preconditions:**

* Grenade has been spawned into scene.

**Postconditions:**

* Thescript is now active.
* grenadeWasPickedUp, thrown, and hasExploded are set to false.
* Set countdown equal to timer, effectively starting the timer countdown. When the timer hits the Grenade will be destroyed over the network and the Explosion will be spawned in the network.

**void Update():**

**Purpose:** called every frame that the GameObject is active in the scene

**Preconditions:**

* Start() has returned

**Postconditions:**

* This has checked whether or not grenadeWasPickedUp is true. If that is true, then the FragGrenade will be destroyed from the map.
* This has checked whether or not playerWhoOwnsThisGrenade is not null and if thrown is false. If these conditions are met, then the FragGrenade’s collider trigger will be turned off, the RigidBody component’s isKinematic variable will be set to false, and FragGrenade.Throw() is called.
* This has checked whether or not thrown is true. If thrown is true, begin to subtract time passed since the last frame was called from the countdown. When the countdown reaches 0, this called Explode() over RPC.

**[PunRPC] void Explode():**

**Purpose:** Instantiates an explosion over the network, and calculates enemies hit within the sphere of the explosion. Applies damage to those enemies based on their xyz proximity to the explosion.

**Preconditions:**

* Timer has reached 0 and the grenade is ready to explode.

**Postconditions:**

* Explosion has been spawned.
* Damage and physics force has been applied to all targets within the explosion radius.

**[PunRPC] void DestroyRPC():**

**Purpose:** Destroys the GameObject on all clients via RPC.

**Preconditions:**

* DestroyRPC has been called: either a player has picked up a grenade from the arena, or a player has thrown a grenade and it has just exploded.

**Postconditions:**

* This has checked whether or not the device is the master client. If it is the master client, it will destroy the GameObject, sending the RPC call to all remote clients.

**void Throw():**

**Purpose:** Applies force to the grenade so that it can appear to be lobbed.

**Preconditions:**

* playerWhoOwnsThisGrenade is set to the MonoBehaviourPun that belongs to the player who owns the grenade, and thrown is false (the grenade has not yet been thrown).

**Postconditions:**

* Set thrown equal to true.
* Physics force has been applied to the rigidbody, and the grenade is flying through the air.

### FragExplosion Class

**Class Purpose**

Represents the explosion that is instantiated by throwing a frag grenade.

**Data Fields**

**public AudioClip explosionSound:** The storage location for a wav file that plays the explosion sound.

**AudioSource audioSource:** the AudioSource component that plays the AudioClip

**Methods**

**void Start():**

**Purpose:**

* Called once and only once at the beginning of the script’s lifecycle when the associated GameObject is instantiated.
* Used to set values before Update() begins.
* Thescript is now active.

**Preconditions:**

* Grenade has been thrown.
* Explosion has been instantiated.

**Postconditions:**

* audioSource is set to the Explosion’s AudioSource component.
* audioSource plays the sound file associated with the Explosion.

**void Update():**

**Purpose:** called every frame that the GameObject is active in the scene

**Preconditions:**

* Start() has returned.

**Postconditions:**

* This has checked whether or not the particle system is still rendering in the scene.
* When the Explosion’s particle system is done rendering, destroy the Explosion GameObject from the scene.

### Gun Class

**Class Purpose**

Describes and manages the gun.

**Data Fields**

**public float damage**: how much damage a gun's bullet can impart (on a target).

**public float range**: how far a bullet can go.

**public float fireRate**: how fast (per second) a bullet can be fired.

**public float impactForce**: force imparted on a bullet hit.

**public GameObject gunPrefab**: Gun Prefab - used to determine type of gun programatically.

**public AudioClip gunshotSound**: Audio Clip (wav file) played when gun is fired

**public ParticleSystem muzzleFlash**: Muzzle flash displayed at the end of the gun when it is fired.

**public GameObject impactEffect**: Visual effect displayed when a bullet hits something.

**public MonoBehaviourPun playerWhoOwnsThisGun**: The player who is holding the gun.

**public Camera fpsCam**: Camera of the player holding the gun. This camera is used for raytracing (determining trajectory of bullet).

**public bool gunShootsItselfImplementation**: Whether the program uses the implementation where the gun is responsible for shooting itself (as opposed to player shooting gun) when user input “Fire1.”

**private AudioSource audioSource:** keeps a reference to the audio source of the gun.

**private float nextTimeToFire:** used to make sure we don't fire faster than fireRate allows.

**Methods**

**public void Start():**

**Purpose:**

* Initializes member variables.

**Preconditions:**

* GameObject associated with this script has been instantiated.

**Postconditions:**

* Member variables are initialized.
* The script is now active.

**public bool IsReadyToShoot():**

**Purpose:**

* Determines whether the gun is ready to shoot.

**Preconditions:**

* Associated GameObject has been instantiated.

**Returns:**

* True if gun is ready to shoot. Otherwise, false

**public int GetTypeOfGun():**

**Purpose:**

* Returns the type of gun

**Preconditions:**

* None

**Returns:**

* Type of gun

**public void Shoot():**

**Purpose:**

* Called by PlayerManager every time the gun needs to be shot. Protects against shooting faster than fire rate allows

**Preconditions:**

* Gun is owned by player

**Postconditions:**

* Gun has shot, precipitating a raycast to simulate the bullet path.
* If a player has been hit, subtract the damage caused from their *Health.*

### Medkit Class

**Class Purpose**

Represents a Medkit that can appear in the arena or in a player’s inventory when picked up from a location in the arena. Players may use the Medkit once picked up in order to regain 30 health.

**Data Fields**

**public float healthThisMedkitWillRestore = 30.0f:** the amount of *Health* that the Medkit is set to restore.

**public bool medkitWasPickedUp:** the boolean that represents whether or not a player has picked up the Medkit from the arena.

**public bool playerUsedMedkit:** the boolean that represents whether or not a player has used the Medkit.

**public float timer = 1.0f:** the time delay it takes for the medkit to be used after the player has pressed.

**public float countdown:** represents the countdown from the timer value to 0.

**public PlayerManager playerWhoOwnsThisMedkit:** a reference to the player who owns this Medkit.

**Methods**

**void Awake():**

**Purpose:**

* Called once and only once at the beginning of the script’s lifecycle. Used to set values before Start() and Update() are called.

**Preconditions:**

* Medkit has been instantiated in arena or used by player.

**Postconditions:**

* playerWhoOwnsThisMedkit set to null.
* playerUsedMedkit set to false.
* Countdown set equal to timer.

**void Update():**

**Purpose:**

* Called every frame that the GameObject is active in the scene.

**Preconditions:**

* Start() has returned.

**Postconditions:**

* This has checked whether or not medkitWasPickedUp is true. If that is true, then the Medkit will be destroyed from the map.
* This has checked whether or not playerWhoOwnsThisMedkit is not null and if playerUsedMedkit is false. If these conditions are met, then the Medkit’s collider trigger will be turned off and Medkit.Use() is called.
* This has checked whether or not playerUsedMedkit is true. If it is true, begin to subtract time passed since the last frame was called from the countdown. When the countdown reaches 0 (from 1 second), this calls RestoreHealth on the client.

**void RestoreHealth():**

**Purpose:** Medkit wrapper class for the PlayerManager.RestoreHealth method.

**Preconditions:**

* Player has pressed H on the keyboard, and PlayerManager.ProcessInputs() has received the input keycode.
* Player does not have maximum health.

**Postconditions:**

* Medkit spawned and deactivated very quickly, since it is not necessary to see the medkit box GameObject when using it.
* Player has regained 30 health.
* If player has more than the max health, set their health to the max health (this logic is handled in PlayerManager, but remains relevant enough to mention here).

**[PunRPC] void DestroyRPC():**

**Purpose:**

* Destroys the GameObject on all clients via RPC.

**Preconditions:**

* DestroyRPC has been called: either a player has picked up a medkit from the arena.

**Postconditions:**

* This has checked whether or not the device is the master client. If it is the master client, it will destroy the GameObject, sending the RPC call to all remote clients.

**void Use():**

**Purpose:**

* Function called when player uses a medkit.

**Preconditions:**

* playerWhoOwnsThisMedkit that is set to the PlayerManager, which represents the owner of the medkit, is not null, and playerUsedMedkit is false.

**Postconditions:**

* Set playerUsedMedkit equal to true, which sets the condition to precipitate player to regain 30 health.

### Chat Manager Class

**Class Purpose**

**Provides an interface through which players within a game can communicate with each other**

**Data Fields:**

**private PhotonView PV:** This object’s photon view

**public GamePlayFabController GPFC:** The singleton that handles database methods

**public GameObject CB:** The chat box

**public string username:** The user’s username

**public int maxMessages:** The max messages that can be stored in the scroll view

**public bool setToDisable:** Bool for chat disabling

**public GameObject chatPanel:** The chat UI

**public GameObject textObject:** Text message

**public InputField chatBox:** The input field

**public Color playerMessage:** Color for player messages

**public Color info:** Color for system messages

**List<Message> messageList:** List of messages

**Methods**

**public void OnEnable():**

**Purpose:**

* Establish a Photon callback target.

**Preconditions:**

* **GameObject is instantiated**

**Postconditions:**

* **ChatManager has a Photon callback target.**

**public void OnDisable():**

**Purpose:**

* **Dispose of resources when chat not in use.**

**Preconditions:**

* **ChatManager going from enabled state to disabled state.**

**Postconditions:**

* **ChatManager Photon callback target is disposed of.**

**public void Start():**

**Purpose:**

* **Assigns PhotonView to the ChatManager.**

**Preconditions:**

* **ChatManager is instantiated.**

**Postconditions:**

* **ChatManager has PhotonView component.**

**public void Update():**

**Purpose:**

* **KeyCodes are interpreted, chat box activation and deactivation is processed, messages are sent from input field of the chat box to the scroll view of the chat box via RPC call.**

**Preconditions:**

* **ChatManager is instantiated with PhotonView.**
* **Start() has returned.**

**Postconditions:**

* **Messages stored in the input field of a user’s chat box are placed in the scroll view. System messages and player messages are color coded.**

**public void RPC\_SendMessageToChat(string text, Message.MessageType messageType)**

**Purpose:**

* **Transmit chat messages from one user over the network for all users to see.**

**Preconditions:**

* **User has entered some text in the chat box input field and hit enter.**

**Postconditions:**

* **Message is transmitted over the network and each user can see its contents**.

### CountdownTimer Class

**Class Purpose**

Keeps track of the countdown timers for the game

**Data Fields**

**private bool isTimer1Running:** keeps track of whether timer 1 is running.

**private bool isTimer2Running:** keeps track of whether timer 2 is running.

**private float startTime1:** keeps track of the start time for timer 1.

**private float startTime2:** keeps track of the start time for timer 2.

**Methods**

**public void CheckForTimerInfo():**

**Purpose:**

* Checks for timer info in Room.CustomProperties.

**Preconditions:**

* None.

**Postconditions:**

* Timer information is updated.

**public void Start():**

**Purpose:**

* Initializations before Update() has begun.

**Preconditions:**

* GameObject associated with this script has been instantiated.

**Postconditions:**

* Thescript is now active.
* CheckForTimerInfo is called.

**public void Update():**

**Purpose:**

* Updates countdown timers and triggers callbacks.

**Preconditions:**

* Start() has returned.
* Countdown timers have started.

**Postconditions:**

* Countdown timers have updated and callbacks are triggered if necessary.

### JumboTronDisplay Class

**Class Purpose:**

Updates the info on the JumboTron display.

**Methods**

**public void Update():**

**Purpose:**

* Called every frame.
* Updates the info on the JumboTron display.

**Preconditions:**

* Start() has returned.
* One of the two timers is running.

**Postconditions:**

* Timer and stage info is updated on JumboTron display.

### **MobileButtonController Class**

**Class Purpose**

Handles conversion of mobile input into CrossPlatformInput.

**Data Fields**

**bool isChatOpen:** Flags current state of Chat CrossPlatformInput.

**bool isStatsOpen:** Flags current state of Stats CrossPlatformInput.

**Methods**

**public void ToggleChat():**

**Purpose: Toggles chat crossplatforminput.**

**public void ToggleStats():**

**Purpose: Toggles stats crossplatforminput.**

**public void Jump():**

**Purpose: Presses Jump crossplatforminput.**

**public void UseHealth():**

**Purpose: Presses health crossplatforminput.**

**public void ThrowGrenade():**

**Purpose: Presses grenade crossplatforminput.**

**public void ToggleAi():**

**Purpose: Presses Aicrossplatforminput.**

**public void CycleGun():**

**Purpose: Presses Cycle Gun crossplatforminput.**

### PlayerAnimatorManager Class

**Class Purpose**

Manages player’s animator.

**Data Fields**

**[SerializeField] private float directionDampTime**: time to take to smooth out animation.

**Animator animator:** keeps a reference to the player’s animator.

**Methods**

**public void Update():**

**Purpose:**

* Updates player’s animator.

**Preconditions:**

* Player is instantiated on network.

**Postconditions:**

* Player’s animator is updated.

**public void Start():**

**Purpose:**

* **Initializes member variables.**

**Preconditions:**

* GameObject associated with this script has been instantiated.

**Postconditions:**

* Member variables are initialized.

### PlayerNetworkMover Class

**Class Purpose:**

Rotates camera position over the network.

**Data Fields**

**[SerializeField] Camera playerCamera:** Player camera whose vertical rotation we want to sync over network.

**private Quaternion rotation:** holds camera rotation info from network.

**private readonly float smoothing:** Smooths out motion.

**Methods**

**public void Update():**

**Purpose:**

* Changes rotation of camera smoothly.

**Preconditions:**

* Player is instantiated and has an active camera.

**Postconditions:**

* Player’s camera is rotated.

**public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)**

**Purpose:**

* Handles custom synchronization of information over the network.

**Preconditions:**

* Player is instantiated and has an active camera.

**Postconditions:**

* Camera rotation is sent/received over the network.

### WallDropAnimator Class

**Class Purpose**

Manages wall drop animation.

**Data Fields**

**[SerializeField] private float dropTime:** Related to the time it takes for wall to drop the distance equal to its height

**private Vector3 dropPosition:** stores the final position of the wall after it is dropped

**private Vector3 originalWallPosition:** keeps track of the original wall position for ResetWallPosition()

**private float originalY:** keeps track of the original Y position of the wall

**private object[] instantiationData:** holds reference to data that was tied to the instantiation of the object

**private bool wallReachedDropPosition:** keeps track of whether the navmesh is ready to update

**Methods**

**public void Awake():**

**Purpose:**

* Set member variables.

**Preconditions:**

* GameObject associated with this script has been instantiated.

**Postconditions:**

* Member variables are set.

**public void Start():**

**Purpose:**

* Calculates the position to drop the wall.

**Preconditions:**

* Awake() has returned.
* Wall is spawned and in its original position.
* GameObject associated with this script has been instantiated.

**Postconditions:**

* DropPosition is calculated.
* This script is now active.

**public void FixedUpdate():**

**Purpose:**

* Drops the wall when it’s time to do so and triggers an event to indicate when the wall is in its dropped position.

**Preconditions:**

* Wall is spawned.

**Postconditions:**

* Wall is moved/moving towards its appropriate position.

**public void ResetWallPosition():**

**Purpose:**

* Resets the wall position.

**Preconditions:**

* None.

**Postconditions:**

* Wall position is reset.

### WallManager Class

**Class Purpose**

Moves wall’s GameObject to the correct position in the GameObject hierarchy.

**Methods**

**public void Start():**

**Purpose:**

* Moves wall’s GameObject to the correct position in the GameObject hierarchy.

**Preconditions:**

* GameObject associated with this script has been instantiated (wall has been spawned).

**Postconditions:**

* The wall’s GameObject is moved to the correct position in the GameObject hierarchy.
* The script is now active.

### WallTarget Class

**Class Purpose**

Manages the wall’s interactions as a target for gun fire.

**Data Fields**

**[SerializeField] private float teamA\_health: Health for the side of the wall.**

**[SerializeField] private float teamB\_health: health for the side of the wall.**

**[SerializeField] private int wallSide: Wall Side is either 1 or 2.**

**Methods**

**public void Awake():**

**Purpose:**

* **Sets member variables.**

**Preconditions:**

* N**one.**

**Postconditions:**

* **Member variables are set.**

**public override void OnRoomPropertiesUpdate():**

**Purpose:**

* **Syncs the health of the wall over the networked.**

**Preconditions:**

* **None.**

**Postconditions:**

* **Wall health is updated.**

**public void TakeDamage(float amount, PlayerManager player):**

**Purpose:**

* **Wall Target logs who hits it and health will represent the votes needed to take down the wall.**

**Preconditions:**

* **Wall has spawned.**

**Postconditions:**

* **Wall health has updated and is synced.**

**public void ResetHealth():**

**Purpose:**

* **Resets the health of the wall.**

**Preconditions:**

* **Wall is spawned.**

**Postconditions:**

* **Wall health has respawned.**

### WeaponsMenuManager Class

**Class Purpose**

Manages weapons menu for player gui.

**Data Fields**

**public Transform weaponItemPrefab**: Transform object belonging to the weapon item prefab.

**PlayerManager localPlayerPM**: Local player’s PlayerManager instance.

**GameObject activeWeapon**: The player’s active weapon.

**GameObject[] inactiveWeapons**: List of inactive weapons.

**public int HighlightIndex**: Index of highlighted weapon.

**ArrayList weaponItems**: List of weapons.

**bool readyToMoveHighlight**: Keeps track of whether the user is ready to move the highlight.

**Methods**

**public void OnEnable():**

**Purpose:**

* Sets up the script when it is enabled.

**Preconditions:**

* None.

**Postconditions:**

* Set the localPlayerPM variable.

**public void MoveHighlightIndexForward():**

**Purpose:**

* Moves the highlight index forward.

**Preconditions:**

* None.

**Postconditions:**

* Highlight index is moved forward.

**public void MoveHighlightIndexBackward():**

**Purpose:**

* Moves the highlight index backward.

**Preconditions:**

* None.

**Postconditions:**

* Highlight index is moved backward.

**public void UpdateHighlightInMenu():**

**Purpose:**

* Updates highlight in menu.

**Preconditions:**

* Player has opened their weapons menu.

**Postconditions:**

* Weapon in menu has been highlighted.

**public intGetHighlightedGunViewID():**

**Purpose:**

* Finds the Photon View ID of the Gun that is highlighted in the weapons menu, if there is a gun to be highlighted.

**Preconditions:**

* WeaponItems.Count > 0.

**Returns:**

* gunViewID for the selected/highlighted gun.

### EmailInputField Class

**Class Purpose**

Keeps track of email input from user.

**Data Fields**

**const string PREF\_KEY\_EMAIL:** Key for email preference**.**

**Methods**

**public void Start():**

**Purpose:**

* Initializes input field text.

**Preconditions:**

* None.

**Postconditions:**

* Input field text is set.

**public void SetEmail(string value):**

**Purpose:**

* Sets email preference.

**Params:**

* The string that represents the Email to be set.

**Postconditions:**

* Email preference is set.

### SpinModels Class

**Class Purpose:**

Script that controls the spinning avatars on the matchmaker menu within the Launcher scene. Designed to make the avatar selection slider area more appealing.

**Data Fields**

**[SerializeField] private float rotationRate = .5f :** rotation rate for the avatars.

**[SerializeField] private GameObject[] avatarModels:** the array storing the avatars that can be selected in the avatar preferences.

**Methods**

**public void Start():**

**Purpose:**

* Called once at the start of the GameObject’s lifecycle in order to provide debug statements for avatarModel array length.

**Preconditions:**

* The GameObject this script is attached to has been instantiated within the scene.

**Postconditions:**

* Thescript is now active.
* Developers have been informed via the console of any of the two very tedious errors to sift through associated with the Unity project inspector.

**public void Update():**

**Purpose:**

* Called once every frame.
* Rotates the avatar models in the GUI about the y axis.

**Preconditions:**

* Start() has returned.

**Postconditions:**

* Adjusted the transform.Rotation attribute of the models in the GUI about the y axis by increments of (360 \* rotationRate \* (time since last frame/call to Update())).

### PlayFabController Class

**Class Purpose**

Provide an interface for dealing with database procedures and storage, as well as authenticating users by reading and storing credentials.

**Data Fields:**

**public static PlayFabController PFC:**

**private string userEmail:**

**private string userPassword:**

**private string username:**

**private string displayName:**

**private string sharedGroupId:**

**private List<FriendInfo> \_friends:**

**private List<string> \_friendStringIDs:**

**private List<string> \_sharedGroupMembers:**

**public string sharedGroupRoom:**

**private bool registering = false**

**private CameraAnimationHandler cameraMover::**

**private SceneManager SM:**

**public GameObject loginAndRegisterPanel**

**public GameObject mainMenuPanel:**

**public GameObject optionsPanel:**

**public GameObject matchmakingPanel:**

**public GameObject friendsPanel:**

**public GameObject messagesPanel:**

**public GameObject loginOptionsPanel:**

**public GameObject registerOptionsPanel:**

**public GameObject newUserButton:**

**public GameObject OpenLoginAgain:**

**public GameObject progressLabel:**

**private Dropdown leaderboardFilterDropdown:**

**private string[] namesOfLeaderboards**

**public int leaderboardFilterIndex:**

**public string selectedLeaderboardFilter:**

**Methods**

**public void OnEnable():**

**Purpose:**

* **To set the PlayFabController as this instance.**

**Preconditions:**

* **PlayFabController GameObject is created.**

**Postconditions:**

* **PlayFabController singleton is instantiated.**

**public void Start():**

**Purpose:**

* **Assign variables and auto-login the user is they have auto-login selected and have player preferences stored locally.**

**Preconditions:**

* **PlayFabController is instantiated.**

**Postconditions:**

* **PlayFabController variables are set and player is auto-logged in if they’ve chosen that option.**

**public void Update():**

**Purpose:**

* **Menu navigation via UI elements is handled.**

**Preconditions:**

* **PlayFabController is instantiated.**

**Postconditions:**

* **None.**

**public void OnClickLogin() :**

**Purpose:**

* **Allow a user to begin the log in process**

**Preconditions:**

* **User has entered data in to the relevant credential fields**

**Postconditions:**

* **If credentials match those present in the database, OnEmailLoginSuccess is called. Otherwise, OnEmailLoginFailure is called**

**private void OnEmailLoginSuccess(LoginResult result)**

**Purpose:**

* **Handles a successful login attempt using email**

**Preconditions:**

* **User has entered appropriate credentials**

**Postconditions:**

* **User is logged in and session token is established**

**private void OnEmailLoginFailure(PlayFabError error)**

**Purpose:**

* **Handles when a user enters bad credentials using email**

**Preconditions:**

* **User enters bad credentials**

**Postconditions:**

* **User is not logged in**

**private void OnPlayfabLoginSuccess(LoginResult result)**

**Purpose:**

* **Handles a successful login attempt without email**

**Preconditions:**

* **User has entered appropriate credentials**

**Postconditions:**

* **User is logged in and session token is established**

**private void OnPlayfabLoginFailure(PlayFabError error)**

**Purpose:**

* **Handles when a user enters bad credentials without email**

**Preconditions:**

* **User enters bad credentials**

**Postconditions:**

* **User is not logged in**

**private void OnLoginMobileSuccess(LoginResult result)**

**Purpose:**

* **Handles a successful login via mobile platform**

**Preconditions:**

* **User has a unique device ID**

**Postconditions:**

* **User is logged in and session token is established**

**private void OnLoginMobileFailure(PlayFabError error)**

**Purpose:**

* **Handles an unsuccessful login via mobile platform**

**Preconditions:**

* **User doesn’t have a unique device ID**

**Postconditions:**

* **User is not logged in**

**public void OnClickOpenLoginAgain()**

**Purpose:**

* **Brings the login page back on screen should a user choose to**

**Preconditions:**

* **User has logged in already and has clicked to go back to the login page**

**Postconditions:**

* **User is brought back to the login page**

**public void OnClickRegister()**

**Purpose:**

* **Handles new users and uses their information to create a new PlayFab account**

**Preconditions:**

* **User enters valid credentials (that abide by rules for credentials)**

**Postconditions:**

* **User credentials are sent to the server**

**private void OnRegisterSuccess(RegisterPlayFabUserResult result)**

**Purpose:**

* **Handles a successful registration attempt**

**Preconditions:**

* **User has entered acceptable credentials to register**

**Postconditions:**

* **A new account for the user is created, and a session token is established**

**private void OnRegisterFailure(PlayFabError error)**

**Purpose:**

* **Handles an unsuccessful registration attempt**

**Preconditions:**

* **User has entered bad credentials or is already a user**

**Postconditions:**

* **A new account is not created**

**void OnDisplayName(UpdateUserTitleDisplayNameResult result)**

**Purpose:**

* **Returns the user’s display name**

**Preconditions:**

* **User has an active PlayFab account**

**Postconditions:**

* **Display name is returned for the user**

**public void SetUserEmail(string emailIn)**

**Purpose:**

* **Setter for the user’s email**

**Preconditions:**

* **none**

**Postconditions:**

* **Sets the user’s email**

**public void SetUserPassword(string passwordIn)**

**Purpose:**

* **Setter for the user’s password**

**Preconditions:**

* **none**

**Postconditions:**

* **Sets the user’s password**

**public void SetUsername(string usernameIn)**

**Purpose:**

* **Setter for the user’s username**

**Preconditions:**

* **none**

**Postconditions:**

* **Sets the user’s username**

**public void SetUserDisplayName(string displayNameIn)**

**Purpose:**

* **Setter for the user’s display name**

**Preconditions:**

* **none**

**Postconditions:**

* **Sets the user’s display name**

**private static string returnMobileID()**

**Purpose:**

* **Return the user’s mobile ID**

**Preconditions:**

* **User has a unique mobile ID**

**Postconditions:**

* **mobile ID for the user is returned**

**public void OnClickNewUser()**

**Purpose:**

* **To highlight the register or login panels depending on a user’s purpose**

**Preconditions:**

* **none**

**Postconditions:**

* **Highlighted panel is toggled**

**private void Defocus(GameObject GO, bool dampenColor)**

**Purpose:**

* **To bring different panels out of focus (deactivate)**

**Preconditions:**

* **GameObject GO (panel) is active**

**Postconditions:**

* **GameObject GO (panel) is deactivated**

**private void Refocus(GameObject GO, bool restoreColor)**

**Purpose:**

* **To bring different panels into focus (activate)**

**Preconditions:**

* **GameObject GO (panel) is deactivated**

**Postconditions:**

* **GameObject GO (panel) is activated**

**public void GetLeaderboard()**

**Purpose:**

* **Return a leaderboard for a stat**

**Preconditions:**

* **Stats exist within the database and at least one user has a score for one**

**Postconditions:**

* **Leaderboard is returned**

**void OnGetLeaderboard(GetLeaderboardResult result)**

**Purpose:**

* **Display Leaderboard in a formatted way**

**Preconditions:**

* **Leaderboard is received from the server**

**Postconditions:**

* **Leaderboard is presented**

**public void CloseLeaderboardPanel()**

**Purpose:**

* **Closes the leaderboard panel**

**Preconditions:**

* **Leaderboard panel is active**

**Postconditions:**

* **Leaderboard panel is deactivated**

**void OnErrorLeaderboard(PlayFabError error)**

**Purpose:**

* **Handles bad leaderboard requests**

**Preconditions:**

* **Leaderboard request is invalid**

**Postconditions:**

* **Nothing returned**

**void AddFriend(FriendIdType idType, string friendId)**

**Purpose:**

* **Adds a friend to the user by searching for their PlayFabID**

**Preconditions:**

* **Friend exists in the database as a user**

**Postconditions:**

* **Friend is either added or not depending on whether or not they exist**

**public void GetFriends()**

**Purpose:**

* **Return an array of the user’s friends**

**Preconditions:**

* **User has friends**

**Postconditions:**

* **Array of friends is returned**

**void DisplayFriends(List<FriendInfo> friendsCache)**

**Purpose:**

* **Present the user’s friends to the user**

**Preconditions:**

* **User has friends**

**Postconditions:**

* **Friends list is presented**

**void DisplayPlayFabError(PlayFabError error)**

**Purpose:**

* **Handles bad display friends requests**

**Preconditions:**

* **User has no friends**

**Postconditions:**

* **No list is returned**

**void DisplayError(string error)**

**Purpose:**

* **Handles bad display requests**

**Preconditions:**

* **There is a network issue upon request**

**Postconditions:**

* **Nothing is returned**

**public void OnClickAddFriend()**

**Purpose:**

* **Calls the AddFriend method**

**Preconditions:**

* **none**

**Postconditions:**

* **AddFriend() is called**

**public void SetFriendUsername(string value)**

**Purpose:**

* **Assigns the friend username to be searched for**

**Preconditions:**

* **none**

**Postconditions:**

* **Friend username is set to the value**

**public void OnClickShowFriendsAndLeaderboard()**

**Purpose:**

* **Brings up the FriendsAndLeaderboard panel**

**Preconditions:**

* **none**

**Postconditions:**

* **Camera is rotated to reveal the panel**

**public void CreateSharedGroup(string Id)**

**Purpose:**

* **Creates a shared group for the user upon registration**

**Preconditions:**

* **User is logged in**

**Postconditions:**

* **User has a shared group assigned to them**

**public void OnCreateSharedGroup(CreateSharedGroupResult result)**

**Purpose:**

* **Assigns a user to their shared group**

**Preconditions:**

* **User successful created a shared group**

**Postconditions:**

* **User is in the shared group they created**

**public void OnErrorCreateSharedGroup(PlayFabError error)**

**Purpose:**

* **Handles bad shared group creation requests**

**Preconditions:**

* **User is already part of the group, or the group name is taken**

**Postconditions:**

* **User is not assigned to any group**

**public void OnClickAddSharedGroupMembers()**

**Purpose:**

* **Handles the adding of users in the database to a shared group**

**Preconditions:**

* **Users exist within the database and aren’t already in the shared group**

**Postconditions:**

* **Users are in the shared group**

**public void OnAddSharedGroupMembers(AddSharedGroupMembersResult result)**

**Purpose:**

* **Handles good AddSharedGroupMember requests**

**Preconditions:**

* **Users are addable to the shared group**

**Postconditions:**

* **Users are added**

**public void OnErrorAddSharedGroupMembers(PlayFabError error)**

**Purpose:**

* **Handles bad AddSharedGroupMember requests**

**Preconditions:**

* **Users aren’t addable to the shared group**

**Postconditions:**

* **Users aren’t added**

**public void FriendsToStringList()**

**Purpose:**

* **Returns the user’s friends as a string List**

**Preconditions:**

* **User has a friends list**

**Postconditions:**

* **User’s friends list is stored as a List of strings**

**public void GetSharedGroupData()**

**Purpose:**

* **Returns data about the user’s shared group**

**Preconditions:**

* **User has a shared group**

**Postconditions:**

* **Data about the shared group is returned**

**public void OnGetSharedGroupData(GetSharedGroupDataResult result)**

**Purpose:**

* **Provides an interface by which a user can send information to their shared group**

**Preconditions:**

* **User has a shared group**

**Postconditions:**

* **Shared group now has access to any information sent by the user**

**public void OnErrorGetSharedGroupData(PlayFabError error)**

**Purpose:**

* **Handles bad SharedGroupData requests**

**Preconditions:**

* **User doesn’t have a shared group or the network is bad**

**Postconditions:**

* **Nothing is returned**

**public void OnClickSetPrivateRoomName()**

**Purpose:**

* **Sets the private room name based on what was set in the shared group data**

**Preconditions:**

* **User is in a shared group and the shared group has a private room name field**

**Postconditions:**

* **Private room is now set in the shared group**

**public void OnUpdateSharedGroupData(UpdateSharedGroupDataResult result)**

**Purpose:**

* **Handles data sent in to the shared group**

**Preconditions:**

* **User has a shared group**

**Postconditions:**

* **Data is stored in the shared group**

**public void OnErrorUpdateSharedGroupData(PlayFabError error)**

**Purpose:**

* **Handles bad shared group data requests**

**Preconditions:**

* **User doesn’t have a shared group or network is bad**

**Postconditions:**

* **No data is added**

**public void SetSharedGroupRoom(string sharedGroupRoomIn)**

**Purpose:**

* **Sets the private room for the shared group**

**Preconditions:**

* **User is part of a shared group**

**Postconditions:**

* **Private room name is set to what the user decides**

**public void SetSharedGroupID(string groupIDIn)**

**Purpose:**

* **Sets the user’s shared group ID**

**Preconditions:**

* **none**

**Postconditions:**

* **User’s shared group ID is set**

**GamePlayFabController Class**

**Class Purpose**

Handles database dealings during gameplay, establishing each user’s username, as well as processing statistic updates

**Data Fields**

**public static GamePlayFabController GPFC:** Singleton for this class

**public SceneManager SM:** Handles the switching of game scenes

**public string username:** The logged in user’s username

**private int playerKillCountThisGame:** The logged in user’s kill count during this game 

**private int playerTotalKills:** The logged in user’s total kills over their account’s history

**private int playerDeathCountThisGame:** The logged in user’s death count during this game

**private int playerTotalDeaths:** The logged in user’s total deaths over their account’s history

**private int playerRoundWinsThisGame:** The logged in user’s round wins during **this** game 

**private int playerTotalRoundWins:** The logged in user’s total round wins over their account’s history

**Methods**

**private void OnEnable()**

**Purpose:** Assigns this instance of GamePlayFabController as a singleton

**Preconditions:** GamePlayFabController GameObject exists

**Postconditions:** GamePlayFabController is created

**public void Start()**

**Purpose:** Sets the Title ID and retrieves the logged in user’s username

**Preconditions:** User is logged in

**Postconditions:** User’s username and Title ID are set

**public void OnGetAccountInfoSuccess(GetAccountInfoResult result)**

**Purpose:** Handles when username retrieval is successful

**Preconditions:** User is logged in

**Postconditions:** User’s username is returned

**void getStats()**

**Purpose:** Retrieves the user’s statistics stored in the database

**Preconditions:** User is logged in and has statistics set in the database

**Postconditions:** User’s statistics are returned

**void OnGetStatistics(GetPlayerStatisticsResult result)**

**Purpose:** Handles a successful attempt to retrieve a user’s stats

**Preconditions:** User is logged in and has stats

**Postconditions:** User’s stats are retrieved

**public void StartCloudUpdatePlayerStats()**

**Purpose:** User’s stats are updated in the database based on events that occur in game

**Preconditions:** User has appropriate stats in the database and their values have changed since the last update

**Postconditions:** Stored stats in the database reflect the user’s actual statistics

**private static void OnCloudUpdateStats(ExecuteCloudScriptResult result)**

**Purpose:** Handles a successful attempt to update a user’s statistics

**Preconditions:** User’s statistics are successfully updated

**Postconditions:** Information regarding the update is returned to the user

**private static void OnErrorShared(PlayFabError error)**

**Purpose:** Handles unsuccessful attempts to update a user’s stats

**Preconditions:** User experiences a connection issue

**Postconditions:** Stats are not updated in the database

**public void IncrementKillCount()**

**Purpose:** Increments the player’s KillCount stat by 1

**Preconditions:** Player successfully kills an opponent in game

**Postconditions:** Player’s new kill count is reflected in the database

**public void IncrementDeathCount()**

**Purpose:** Increments the player’s DeathCount stat by 1

**Preconditions:**  Player is killed by an opponent in game

**Postconditions:** Player’s new death count is reflected in the database

**public void IncrementRoundWins()**

**Purpose:** Increments the player’s RoundWins stat by 1

**Preconditions:** Player is on the winning team of a round of gameplay

**Postconditions:** Player’s new round wins count is reflected in the database

**LeaderboardListing Class**

**Class Purpose**

Provides a prefab for listings within the leaderboard panel for each user/stat combination

**Data Fields:**

### public Text playerNameText: The associated player’s name .

### public Text playerScoreText: The associated player’s score, depending on the leaderboard presented.

### **Launcher** Class

**Class Purpose**

The main menu to launch a game of Armament.

**Data Fields**

**public const string KEY\_ARENA\_FILTER:** connection status displayed to the user.

**private Transform[] spawnPoints:** an array of (x,y,z) coordinates where players will spawn.

**private Camera sceneCamera:** used in Unity to set up a camera within the scene, which is viewed by the user.

**private GameObject[] playerModel:** stores an array of player prefabs (the graphical representations of each player as a 3d model to be spawned in the environment).

**private GameObject serverWindow:** UI panel to display *lobby* info.

**private InputField username:** input field for user to enter username.

**private InputField roomName:** input field for user to enter room name.

**private Text messagesLog:** text field to display messages to the user.

**private GameObject player:** local player instance.

**private Queue<string> messages:** keep track of all messages.

**private const int messageCount:** number of messages in queue.

**private string nickNamePrefKey:** user’s preferred nickname.

**Methods**

**public void Start():**

**Purpose:**

* Called once and only once at the beginning of the script’s lifecycle. Used to set values before Update() begins.

**Preconditions:**

* NetworkManager script is enabled.

**Postconditions:**

* Networking GUI initialized.

**public void JoinRoom():**

**Purpose:**

* Button click callback function for join room.

**Preconditions:**

* Client clicked a button to join room.

**Postconditions:**

* Client has attempted to join room.

**public void Respawn(float spawnTime):**

**Purpose:**

* Start *spawn* or *respawn* a player.

**Params:**

**float spawnTime:** time waited before spawning a player.

**Preconditions:**

* Player is connected and ready to *spawn*.

**Postconditions:**

* Player has *spawned*.

**public void AddMessage():**

**Purpose:**

* Add message to message panel.

**Params:**

**string message:** the message to be added to the panel.

**Preconditions:**

* Message enqueued.

**Postconditions:**

* Message added to GUI panel.

**[PunRPC] void AddMessage\_RPC()**

**Purpose:**

* RPC function to call add message for each client.

**Params:**

**string message:** the message that we want to add.

**Preconditions:**

* Message enqueued.

**Postconditions:**

* Message broadcasted to clients.

**public override void OnConnectedToMaster():**

**Purpose:**

* Callback function called on the client when user has successfully connected to a master Photon server. Once players are connected to the master server, they can join a game lobby.

**Preconditions:**

* Client has connected to a master server.

**Postconditions:**

* N/A

**public override void OnDisconnected(DisconnectCause cause):**

**Purpose:**

* Callback function called on the client when the connection was lost or you disconnected from the server.

**Params:**

**DisconnectCause cause:** Cause of disconnection

**Preconditions:**

* Client is disconnected from server

**Postconditions:**

* Client disconnection response handled

**public override void OnJoinedLobby():**

**Purpose:**

* Callback function when a player joins a lobby.

**Preconditions:**

* Client has joined lobby

**Postconditions:**

* N/A

**public override void OnJoinedRoom**

**Purpose:**

* Callback function on joined room.

**Preconditions:**

* Player has joined lobby.

**Postconditions:**

* Player enters game room.

**public override void OnPlayerLeftRoom()**

**Purpose:**

* Callback function when other player disconnected.

**Preconditions:**

* Player is in game room.

**Postconditions:**

* Player has left game room.

### PlayerManager Class

**Class Purpose**

Manages player information.

**Data Fields**

**private Animator animator:** animator for player 3d object.

**private int[] inventory:** holds the id of each item and weapon held by player.

**private int currentHealth:** stores current health level of a player.

**private bool damaged:** whether the player has been damaged.

**private Image damageImage:** image updated in GUI to signify damage.

**private AudioClip deathClip:** clip that plays once player dies.

**private FirstPersonControl fpController:** controller for first-person player view.

**private Slider healthSlider:** UI slider used to display player health.

**private AudioClip hurtClip:** audio clip that plays when player takes damage.

**private bool isDead:** whether or not a player is dead.

**private AudioSource player:** audio source for player sounds.

**private float respawnTime:** amount of time it takes for player to respawn after death.

**private int startingHealth:** amount of health a player starts with.

**Methods**

**[PunRPC] private void Death(string enemyName):**

**Purpose:**

* RPC function to declare death of player.

**Params:**

**string enemyName:** the name of the enemy that caused player death

**Preconditions:**

* Player is alive with currentHealth <= 0.

**Postconditions:**

* Player is dead, other players are notified.

**private IEnumerator DestroyPlayer(float delayTime):**

**Purpose:**

* Destroy the player game object

**Params:**

**float delayTime:** time in seconds to delay destruction of player

**Preconditions:**

* N/A

**Postconditions:**

* Player is destroyed

**public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info):**

**Purpose:**

* Used to customize synchronization of certain variables across RPC in a script watched by a Photon network view.

**Params:**

**PhotonStream stream:** the network bit stream

**PhotonMessageInfo info:** the network message information

**Preconditions:**

* N/A

**Postconditions:**

* N/A

**private void Start():**

**Purpose:**

* Called once and only once at the beginning of the script’s lifecycle.
* Used to set values before Update() begins.

**Preconditions:**

* Environment initialized.
* The player’s avatar has spawned into the arena.

**Postconditions:**

* ThePlayerManagerscript is now active.

**[PunRPC] public void TakeDamage(int amount, string enemyName):**

**Purpose:**

* Cause player to take damage

**Params:**

**string enemyName:** The name of the enemy that caused damage

**int amount:** The amount of damage caused

**Preconditions:**

* Player is not dead

**Postconditions:**

* Player has taken damage

**public void Update():**

**Purpose:**

* Overridden from class Monobehavior (Unity API). Update is called every frame, if the MonoBehaviour is enabled.

**Preconditions:**

* Player instantiated.

**Postconditions:**

* Player attributes and properties updated.

**public void pickUpItem():**

**Purpose:**

* Player picks up an item that they are near.

**Preconditions:**

* Player in proximity of item.

**Postconditions:**

* Player has a new item in inventory.

### PlayerStats Class

**Class Purpose**

Manages player stats in the current match.

**Data Fields**

**private string name:** stores the player’s username

**private int kills:** tracks the number of kills a player has achieved in a match

**private int deaths:** tracks the number of times a player has died in a match

**Methods**

**public void IncreaseKillCount():**

**Purpose:**

* Increment the kill field.

**Preconditions:**

* The player has killed an opposing player.

**Postconditions:**

* The player’s kill count has increased by one.

**public void IncreaseDeathCount():**

**Purpose:**

* Increment the death field.

**Preconditions:**

* The player has just died.

**Postconditions:**

* The player’s death count has increased by one.

### 

### PlayFabManager Class

**Class Purpose**

Database management, authentication, polling and pushing

**Data Fields**

**private string userEmail:** Holds the user’s email.

**private string userPassword:** Holds the user’s password.

**private string username:** Holds the user’s username.

**Methods**

**public override void OnLoginSuccess(LoginResult result):**

**Purpose:**

* Logs a successful authentication.

**Params:**

**LoginResult result:** result information from login process.

**Preconditions:**

* Connection established.

**Postconditions:**

* User authenticated.

**public override void OnLoginFailure(PlayFabError error):**

**Purpose:**

* Logs an unsuccessful authentication.

**Params:**

**PlayFabError error:** Error specific to login failure.

**Preconditions:**

* Database connection established.

**Postconditions:**

* Authentication failed.

**public override void OnRegisterSuccess(RegisterPlayFabUserResult result)**

**Purpose:**

* Logs a successful registration.

**Params:**

**RegisterPlayFabUserResult result**: Result information from PlayFab user registration.

**Preconditions:**

* Connection established.

**Postconditions:**

* User account created and user authenticated.

**public override void OnRegisterFailure(PlayFabError error )**

**Purpose:**

* Logs an unsuccessful authentication.

**Params:**

**PlayFabError error:** Error information from user registration failure

**Preconditions:**

* Connection established.

**Postconditions:**

* Register failed.

**public string GetUserEmail():**

**Purpose:**

* Returns the user’s email.

**Preconditions:**

* Connection established.

**Postconditions:**

* Email is returned.

**public string GetUsername():**

**Purpose:**

* Returns the user’s username.

**Preconditions:**

* Connection established.

**Postconditions:**

* Username is returned.

### 

### IShootable Interface

**Interface Purpose**

Interface for all gun objects.

**Methods**

**public void Throw():**

**Purpose:**

* The method that applies the throwing force to the throwable GameObject.

**Preconditions:**

* N/A

**Postconditions:**

* N/A

**public void ISReadyToThrow():**

**Purpose:**

* The method that checks whether the *cooldown* on the throwable GameObject has reset.

**Preconditions:**

* N/A

**Postconditions:**

* N/A

### IThrowable Interface

**Interface Purpose**

Interface for all useable item objects (non-weapon).

**Data Fields**

**private string name:** character representation of the name of the item.

**private string description:** description of the item.

**private int id:** unique id for item.

**Methods**

**public void Throw():**

**Purpose:** allows the user to throw the grenade they picked up.

**Preconditions:** player has grenade in grenade UI slot (far right panel)..

**Postconditions:** player no longer has grenade in inventory, and the grenade will be propelled through the air, and cause damage within a spherical radius. The amount of damage is determined by how close players are to the explosion using an xyz distance equation.

### IUseable Interface

**Interface Purpose**

Interface for all useable item objects (non-weapon).

**Data Fields**

**private string name:** character representation of the name of the item.

**private string description:** description of the item.

**private int id:** unique id for item.

**Methods**

**public void Use():**

**Purpose:** allows the user to use the medkit they picked up.

**Preconditions:** player has item in medkit UI slot (far right panel).

**Postconditions:** player no longer has medkit in inventory, and player recovers 30 health.