**Test Report**

**<ARMAMENT>**

****

**REVISION HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| Revision # | Author | Revision Date | Comments |
| 1.0 | Keith Bosworth | April 28, 2019 | initiated |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[**System Overview**](#_30j0zll) **4**

**Glossary 9**

[**Document Overview**](#_1fob9te) **9**

[**Unit Tests**](#_tyjcwt) **9**

[**Integration Tests**](#_z337ya) **19**

[**Acceptance Tests**](#_2xcytpi) **19**

[**References**](#_147n2zr) **21**

## System Overview

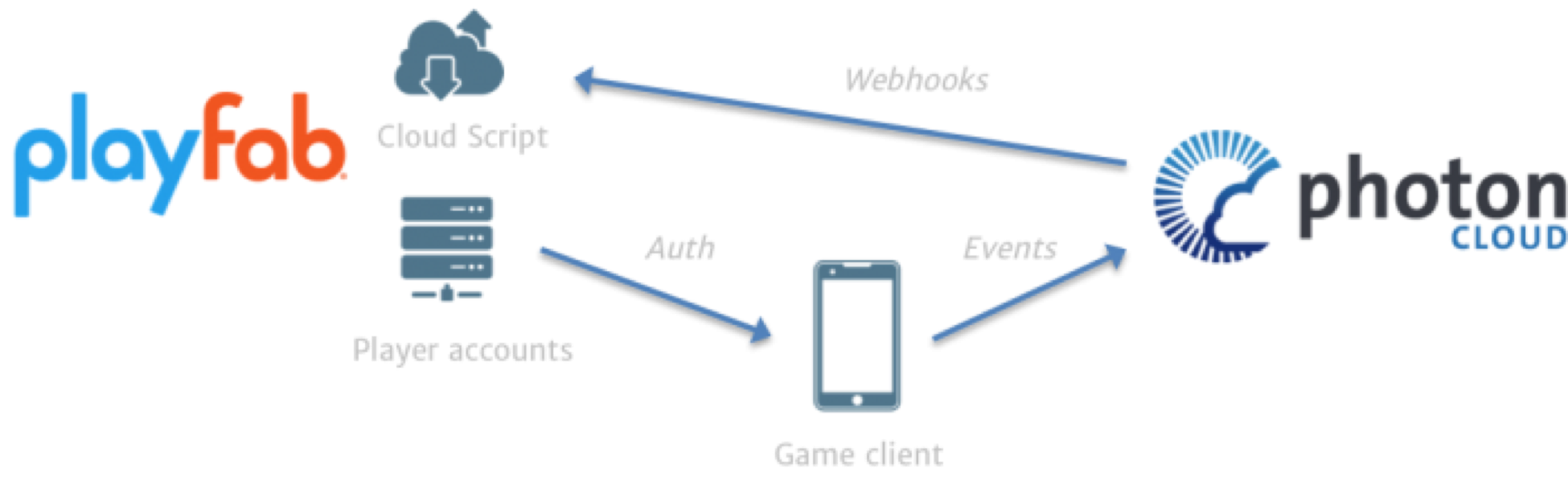
This section provides an overview of the Armament project: an original, networked, multiplayer, team-based video game built for PC, Mac, iOS, and Android platforms. Armament is built with the Unity game engine and API, Photon Networking servers and API, and PlayFab backend database and API.

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.

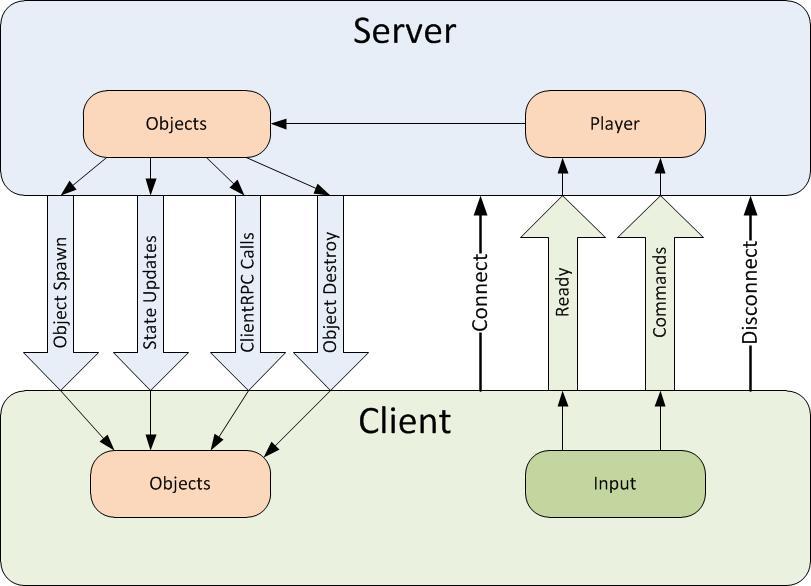
The interaction between Unity clients, the Photon cloud, and PlayFab can be seen below in the abstraction provided by **Figure 1.**

****

**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.

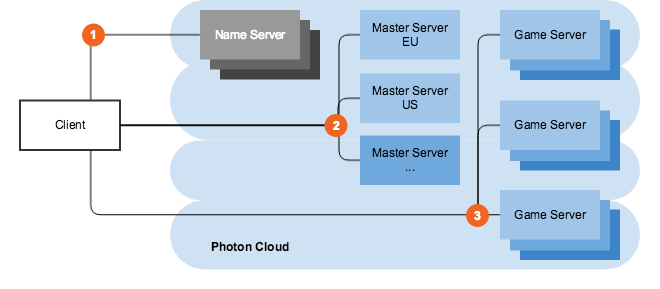
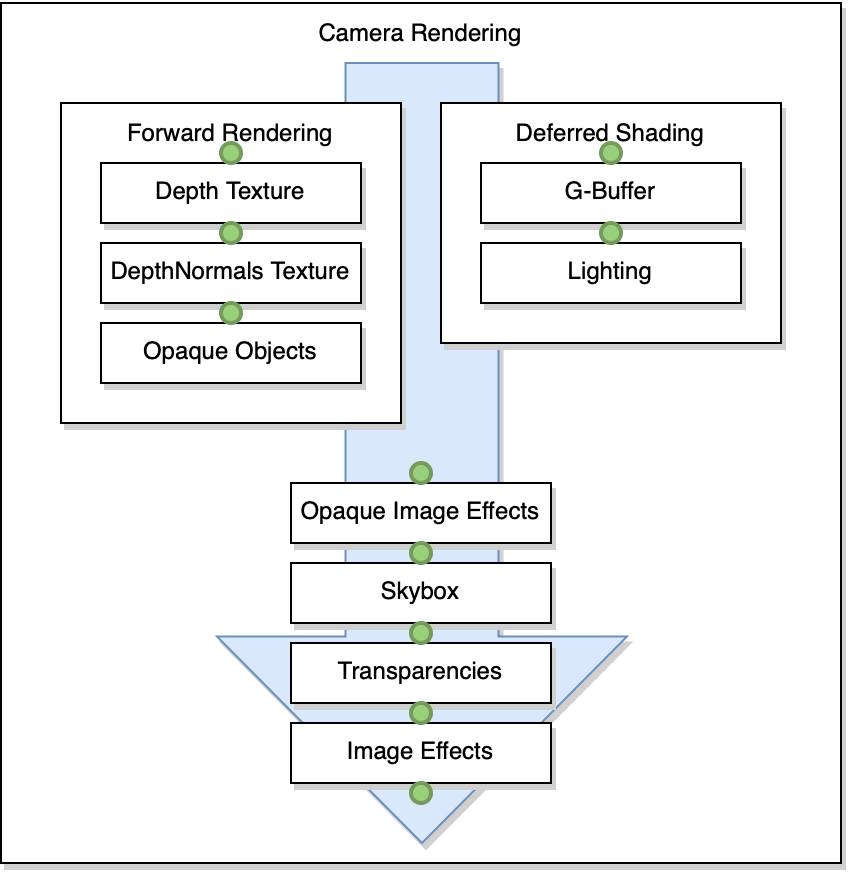


Figure 4.Master server and game server layout..

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.

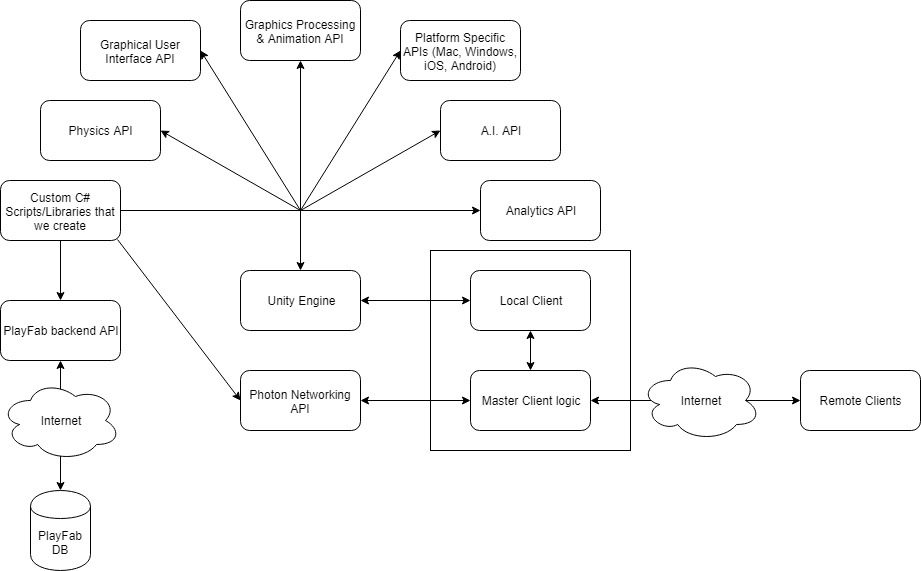
## The onus of graphics processing is placed on the client devices, which may flex the master client’s processing capabilities somewhat, but overall is not an impediment considering how picky our master client is in terms of choosing what to broadcast to clients through Photon servers. On the client 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.
* **GameObject**:
* **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.

## Document Overview

The test report document outlines the various tests completed to verify the extent of completion for requirements applied to the various features developed for Armament. These tests are also used to determine the stability of the overall system. The tests described in this document were defined in the accompanying test procedures document.

## 

## Unit Tests

### GameManager Class Methods

#### Private Methods

* void UpdateNavMesh()
  + Test 1:
    - Outcome: Success
* void LoadArena()
  + Test 1:
    - Outcome: Success
* void UpdatePlayerPropertiesDisplay()
  + Test 1:
    - Outcome: Success
* void RemoveUnclaimedItems()
  + Test 1:
    - Outcome: Success
* void ReturnVanishedItems()
  + Test 1:
    - Outcome: Success
* void EndRound()
  + Test 1:
    - Outcome: Success
* void DestroyAllItems()
  + Test 1:
    - Outcome: Success
* void BalanceTeams()
  + Test 1:
    - Outcome: Success
* void StartRound()
  + Test 1:
    - Outcome: Success
* void SpawnNewItems()
  + Test 1:
    - Outcome: Success
* void SpawnWall()
  + Test 1:
    - Outcome: Success
* GameObject InstantiatePlayerForActor()
  + Test 1:
    - Outcome: Success
* void RemoveGunOwnerships(Player player)
  + Test 1:
    - Outcome: Success

### Gun Class Methods

#### Public Methods

* public void Shoot()
  + Test 1:
    - Outcome: Success
* public void PlayGunShotSound()
  + Test 1:
    - Outcome: Success
* public bool IsReadyToShoot()
  + Test 1:
    - Outcome: Success
* public int GetTypeOfGun
  + Test 1:
    - Outcome: Success

### FragGrenade Class Methods

#### Public Methods.

* [PunRPC] void Explode()
  + Test 1:
    - Outcome: Success
* [PunRPC] void DestroyRPC()
  + Test 1:
    - Outcome: Success
* void Throw()
  + Test 1:
    - Outcome: Success

### Medkit Class Methods

#### Public Methods

* **void RestoreHealth():**
  + Test 1:
    - Outcome: Success
* **[PunRPC] void DestroyRPC():**
  + Test 1:
    - Outcome: Success
* **void Use():**
  + Test 1:
    - Outcome: Success

### Launcher Class Methods

#### Public Methods

* public int GetAvatarSliderValueFromSlider()
  + Test 1:
    - Outcome: Success
* public void Connect()
  + Test 1:
    - Outcome: Success
* public void JoinRandomRoom()
  + Test 1:
    - Outcome: Success
* public void JoinOrCreateRoom()
  + Test 1:
    - Outcome: Success
* public void JoinOrCreateRoom(string[] expectedUsers)
  + Test 1:
    - Outcome: Success

### PlayerManager Class Methods

#### Public Methods

* public void CallShootRPC()
  + Test 1:
    - Outcome: Success
* public void ToggleAIControl()
  + Test 1:
    - Outcome: Success
  + Test 2:
    - Outcome: Success
* public void PlaySound (AudioClip audioclip)
  + Test 1:
    - Outcome: Success
* public void SetTeam(string team)
  + Test 1:
    - Outcome: Success
* public string GetTeam()
  + Test 1:
    - Outcome: Success
* public void TakeDamage(float amount)
  + Test 1:
    - Outcome: Success
* public void TakeDamage(float amount, PlayerManager playerWhoCausedDamage)
  + Test 1:
    - Outcome: Success
* public void RestoreHealth(float amount)
  + Test 1:
    - Outcome: Success
* public void ResetHealth()
  + Test 1:
    - Outcome: Success
* public void ResetShield()
  + Test 1:
    - Outcome: Success
* public void MovePlayer(Transform t)
  + Test 1:
    - Outcome: Success
* public void Respawn()
  + Test 1:
    - Outcome: Success
* public void StartDeadSpectatorMode()
  + Test 1:
    - Outcome: Success
* public void StopDeadSpectatorMode()
  + Test 1:
    - Outcome: Success
* public void SetActiveGun(int gunViewID)
  + Test 1:
    - Outcome: Success
* public void PickUpGun(int gunViewID)
  + Test 1:
    - Outcome: Success
* public void DropAllItems
  + Test 1:
    - Outcome: Success

#### Private Methods

* void DropGun(Gun gun)
  + Test 1:
    - Outcome: Success
  + Test 2:
    - Outcome: Success
* void AddDeath()
  + Test 1
    - Outcome: Success
* void AddKill()
  + Test 1
    - Outcome: Success
* void SetMode(string modeValue)
  + Test 1:
    - Outcome: Success

### WallDropAnimator Class Methods

#### Public Methods

* public void ResetWallPosition()
  + Test 1:
    - Outcome: Success

### CameraAnimationHandler Class Methods

#### Public Methods

* void MoveLeft()
  + Test 1
    - Outcome: Success
* void MoveRight()
  + Test 1
    - Outcome: Success
* void MoveBack()
  + Test 1
    - Outcome: Success
* void MoveUp()
  + Test 1
    - Outcome: Success
* void MoveDown()
  + Test 1
    - Outcome: Success
* void ResetTriggers()
  + Test 1
    - Outcome: Success

### MobileButtonController Class Methods

#### Public Methods

* void ToggleChat()
  + Test 1
    - Outcome: Success
  + Test 2
    - Outcome: Success
* void ToggleStats()
  + Test 1
    - Outcome: Success
  + Test 2
    - Outcome: Success
* void Jump()
  + Test 1
    - Outcome: Success
* void UseHealth()
  + Test 1
    - Outcome: Success
* void ThrowGrenade()
  + Test 1
    - Outcome: Success
* void ToggleAi()
  + Test 1
    - Outcome: Success
* void CycleGun()
  + Test 1
    - Outcome: Success

### GamePlayFabController Class Methods

#### Public Methods

* void OnGetAccountInfoSuccess(GetAccountInfoResult result)
  + Test 1
    - Outcome: Success
* void getStats()
  + Test 1
    - Outcome: Success
* void OnGetStatistics(GetPlayerStatisticsResult result)
  + Test 1
    - Outcome: Success
* void IncrementKillCount()
  + Test 1
    - Outcome: Success
* void IncrementDeathCount()
  + Test 1
    - Outcome: Success
* void IncrementRoundWins()
  + Test 1
    - Outcome: Success

### AICharacterController Class Methods

#### Public Methods

* public void SetTarget(Transform target)
  + Test 1
    - Outcome: Success

#### Private Methods

* void CreateDistanceGraph()
  + Test 1
    - Outcome: Success
* Transform FindNextPointToTarget()
  + Test 1
    - Outcome: Success
* bool EnemyIsInCrosshairs(out PlayerManager enemy)
  + Test 1
    - Outcome: Success
* void ShootGun()
  + Test 1
    - Outcome: Success

### CountdownTimer Class Methods

#### Private Methods

* void CheckForTimerInfo()
  + Test 1
    - Outcome: Success
  + Test 2
    - Outcome: Success

## Integration Tests

#### PhotonIntegrationScript Methods

* TestPhotonConnection()
  + Outcome: Success
* TestPlayerPropertySync()
  + Outcome: Success

#### PlayfabIntegrationScript Methods

* TestPlayfabConnection()
  + Outcome: Success
* TestAccountCreation()
  + Outcome: Success

## Acceptance Tests

* TC0001
  + Outcome: Success
* TC0002
  + Outcome: Success
* TC0004
  + Outcome: Success
* TC0005
  + Outcome: Success
* TC0007
  + Outcome: Success
* TC0008
  + Outcome: Success
* TC0009
  + Outcome: Success
* TC0010
  + Outcome: Success
* TC0011
  + Outcome: Success
* TC0012
  + Outcome: Success
* TC0013
  + Outcome: Success
* TC0014
  + Outcome: Success
* TC0015
  + Outcome: Success
* TC0017
  + Outcome: Success
* TC0018
  + Outcome: Success
* TC0019
  + Outcome: Success
* TC0020
  + Outcome: Success
* TC0021
  + Outcome: Success
* TC0022
  + Outcome: Success
* TC0023
  + Outcome: Success

## References

* Unity user Manual 2018.3 <https://docs.unity3d.com/Manual/index.html>
* Photon documentation <https://doc.photonengine.com/en-us/pun/v2/getting-started/pun-intro>
* Playfab documentation <https://api.playfab.com/docs/general-getting-started>