Multiplayer Online War Simulator Based

On Unity3D

By

Shameer Azmi

Fahim Bakhtiar

Afsana Hossain

A thesis submitted to the faculty of BRAC University in partial fulfillment of the requirements for the degree of Bachelors of Science in Computer Science and Engineering

School of Engineering & Computer Science

Department of Computer Science & Engineering

BRAC University

Declaration

We hereby declare that this is a projects based thesis on the arena of networking and graphics programming. We have gone through numerous reference papers from national and international researchers. Materials of work found by other researchers are mentioned by reference. We have added all the necessary information, and figures in this paper.

Signature of the Supervisor

_____

Professor Dr. Md. Haider Ali

Signature of the Authors

_____

Shameer Azmi

_____

Fahim Bakhtiar

_____

Afsana Hossain

FINAL READING APPROVAL

I have read the thesis on "Multiplayer Online War Simulator Based on Unity3D and OpenGL" by Shameer Azmi, Fahim Bakhtiar and Afsana Hossain in its final form and have found that its format, citations, and bibliography are acceptable. The materials including figures, tables, and charts are in place and the final result is satisfactory. The thesis has been prepared under my Supervision and is a record of bona-fide work carried out successfully.

Approved by

_____

Supervisor
Professor Dr. Md. Haider Ali

Acknowledgement

We are glad to thank our thesis supervisor and our respected chairperson Professor Dr. Md. Haider Ali who spent his valuable time to supervise our thesis work. We are thankful to him for his support and encouragement over the past year. We had major issues with the project such as building a networked domain for multiplayer simulation, latency issues, performance measurement etc. Our Respected advisor Professor Dr. Md. Haider Ali has showed us ways and helped us to solve these problems. We are grateful and humbled by the Department of Computer Science & Engineering, HR Department, and Office of Career Services and Alumni Relations (OCSAR), of BRAC University for providing us confidential data to train our system. Lastly, we are thankful to our family, friends and everyone who helped for their endless support and love.

Abstract

The advent of technology has brought up the need of simulators in various fields especially in combat training programs. Our project will enable to create an environment to interact in such programs. We aim to develop a three dimensional war simulator where multiple players can fight with each other and interact with the environment in real time. The environment will consist of various elements including a terrain to match the need of the simulation with proper obstacles plus natural lighting and it will act as accurately as possible upon interaction to simulate an actual war situation. The players will be able to control their movement besides weapons and based on their command the position of each player as well as their interaction will change. This change of position will result in the change of shadow, lighting and other related elements in the environment & other players will be seeing the changes related to the players in movement.

CONTENTS

Chapter 1
Introduction

War simulations are widely used war games nowadays by governments both individually and collaboratively but little is known about it outside professional circles. Firstly, War Simulations are theories of warfare which can be tested and refined without the need for actual hostilities. In order to provide a better understanding of the way a battle develops into either a victory or a loss, simulations are often used. In fact, the military academies usually use them as tools in the classroom to aid the instruction of tactics and battlefield theory as these simulations provide practical knowledge which cannot be provided with lectures (Brown, 1999). Simulation technologies are now considered as one of the most important ingredients to fuel major strategic capability for soldiers in defense. Every year, developed countries such as U.S.A, China, and South Korea etc. spend lots of money for its defense system. For instance, According to 2015's budget of U.S.A, it has spent $598.5 billion (Military Spending in the United States, n.d.).However, this type of simulation software are very much expensive and out of the reach for a 3rd world developing country like Bangladesh. On the other hand, a world class well-trained and capable defense system is necessary for every country. Keeping that in mind, here we have developed a prototype multiplayer based war simulator which can be used to train our soldiers in near future given the appropriate logistics and budget, thus enabling us to be self sufficient with our own technology and reducing the immense cost on simulators build by others.At the same time it will also create a path to a whole world of simulation software development which were unknown to many of us till date

.

1.1 Motivation:

Combat simulation systems are widely used in military academies for training purposes and as helpful as they are, they also come with a very high cost associated with them. Our motivation roots from the need of a war simulation software in our national defense Purposes. Developing such simulator in our own country would provide the opportunity to manage financial cost for military training and combat simulations. Developing High end latency optimized networking model for 3D virtual space is a necessary demand for simulators as such, which is in fact one of our primary motivation as well.

1.2 Goals:

Our primary goal is to develop an efficient combat simulator with optimized performance. To achieve that we intend to implement a terrain with real life obstacles plus environment and introduce a character which can interact with the environment and also be affected by it. Our aim is to utilize the massive library of unity3d and reduce the overhead of the simulator as it generally tends to slow down when multiple players are introduced in any simulators. We will reduce the latency level to make the simulation to` work more efficiently.

1.3 Methodology and Approaches:

The primary methodology we followed as the foundation of the system is based on client server model. Multiple players will be connected to a centralized server which will stream the game to each of the clients. There will also be a database for storing the information about the environmental and player data. Our selected technologies for the project is, Unity3d as our game engine, Photon Unity Network Manger as the client server system for the multiplayer model, Blender as the 3D modeling software and C# as the programming language.

1.3 Literature review:

Perla in his book (the art of war gaming) said that players in professional war games must understand that their decision making process are the key subjects of game instruction as it is played to meet formal education or training objectives. He also added that effective play of professional war-games requires that the participants prepare for their roles by understanding the real world problems .However, he also added that it is an imperfect mirror of reality that can be utilized by the decision making process of the player. As a result interrupting the insights derived from war gaming requires special care. Regarding the implementation of multiplayer mode, Eric Croninj & Burton Filsturp said that, peer to peer architectures raise a whole set of new issues, the most immediate being game state synchronization, so as opposed to that, choosing a client server model will work better for our simulation program.

Chapter 2

Relevant Work in the Field Of Combat Simulators

2.1 BCS3 (logistical map-based operational simulator):

BCS3 or The Battle Command Sustainment Support System is a real-time logistical map-based operational simulator that has the capabilities to commanders at all echelons, and includes a logistics reporting tool (LRT) for sustainment status reports, supply and equipment in-transit visibility (ITV), and resource (personnel) asset visibility. It is one of the logistics tools within Mission Command and provides sustainment information in the Command Post Computing Environment (CP CE). The CP CE is one of several computing environments nested inside the Army's common operating environment, and aims to simplify systems architecture for command-and-control capability development at tactical echelons.

"Simulations are used to train commanders and staffs in procedures and a wide range of offensive and defensive operations without having to deploy," said Robert Sears, senior subject matter expert for Sustainment System Mission Command.

Exercises are tailored to unit requests, generally last 10 days and include participation from the active Army, Reserve component and the National Guard. The BCS3 architecture allows an organization to execute a myriad of sustainment operations. In more complex situations, the BCS3 architecture can be expanded to accommodate a greater number of systems, sometimes called BCS3 clients.

To ensure the most realistic experience for sustainment support soldiers, the BCS3 Sim-Stim team stimulates the scenarios by injecting data. The injected data test the responses of sustainment personnel. The types of simulations exercised are part of two well-known logistics federations: the Multi-Resolution Federation (MRF) and the Entity Resolution Federation (ERF). MRF uses war fighter simulation (WARSIM) for the combat model and Logistics Federation (LOGFED) for the logistics model. ERF interfaces to Mission Command Systems via the Joint Conflict and Tactical Simulation as the combat model and Joint Deployment Logistics Model as the logistics model.

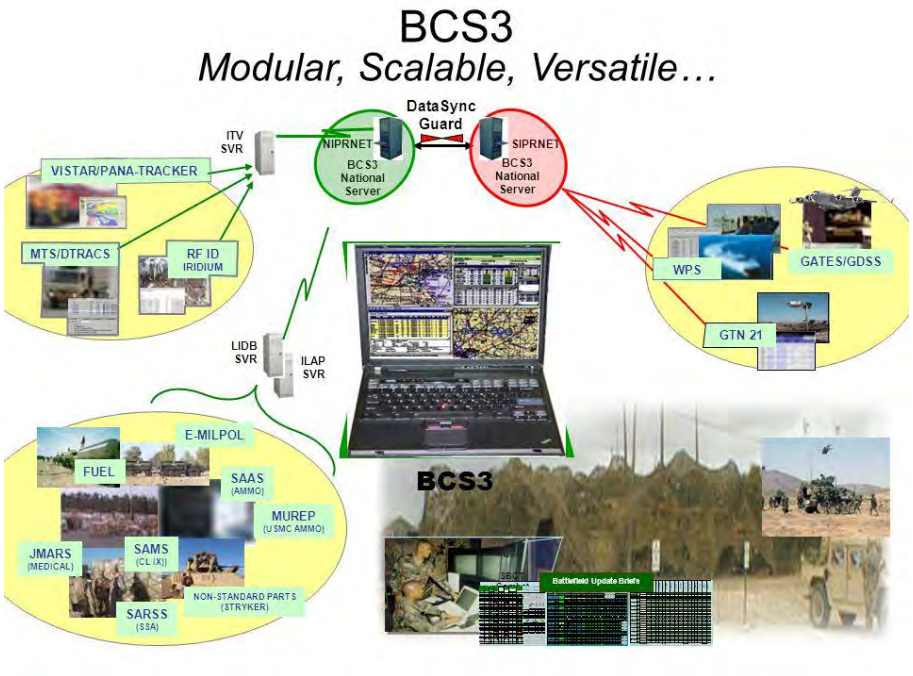

Fig 2.1: The Battle Command Sustainment Support System

The actual scenarios vary depending on the training requirements, so the BCS3 Sim-Stim team closely coordinates each exercise to ensure the exercise meets the command's expectations. For example, a command may state that it needs to prepare for an upcoming deployment and want their soldiers to learn how to track Class I items (food, water and other comfort items). The

simulation then pushes out a supply status to the user's BCS3 terminal, but withholds Class I updates.

The simulated environment begins when logistics data flows from the LOGFED (logistics) server through a gateway to the LOGFED Sim- Stim client, which then feeds to the BCS3 main gateway. The gateway pushes that data down to the BCS3 clients. It is in the Sim-Stim client where the unit task organization and tracked items list, a crucial listing of the commodities commanders deem necessary to complete their mission, are built and passed to the BCS3 NDP and the training BCS3 clients (training audience) via the main gateway.

2.2 Applying The Technology of Distributed Training Simulations to Gaming:

Larry Mellon, Jesse Aronson, Darrin West in their work talks about applying distributed training simulations in gaming. The work is not directly related to combat simulators but the topic included in their work heavily depends on the technology to achieve a successful combat simulator is wide scale.

The U.S. government has invested billions of dollars in simulation-based training and continues to develop the area. Many of these training systems bear a remarkable resemblance to gaming: strategy, tactics, planning and real time combat are all practiced via simulated forces and synthetic immersive battlefields. Increased combat readiness at a lower cost is the answer. The constant exercising of combat tactics under realistic conditions improves response time and allows exploring new tactics for a changing opposition in new situations. Simulation and immersive trainers provide a reasonably realistic environment at a fraction of the cost of war

games in the field. And of course strategy sessions and planning via abstract modeling go back to Alexander drawing options in the sand.

Many of the technical difficulties facing game developers today have been encountered in one form or another by the military training community. As computational power, network connectivity and high-end graphics transition into the low-cost PC market, game developers have the ability to produce a richly textured shared virtual world affordable to a broad base of consumers. Exploiting the potential of the Internet via approaches from distributed interactive training systems is a natural place to begin.

Fig 2.2: Applying Simulation Programs with proper equipments and settings

2.2.1 Classes of Military Simulation:

Distributed simulation is used for an incredibly broad range of purposes within the DoD. A great deal of modeling is done to examine the behavior of potential new military systems under various conditions before construction, and simulation is used to stress-test components during design and construction. The level of detail (or *fidelity*) used in these models is well in excess of gaming requirements: little of value to the gaming community exists in these systems. Other uses of simulation bear a much closer resemblance to gaming:

2.2.2 Real time training:

Operators of military systems are stimulated with inputs similar to combat conditions and respond using simulated control systems. Tank drivers, jet pilots, helicopter jockeys, and fire control center specialists all hone their battle skills in immersive environments populated with both friendly and opposition forces. Fast-twitch action dominates.

2.2.3 Strategic analysis:

Commanders from platoon leaders to generals plan mock battles, analyzing strategic options and potential enemy responses using abstract models of logistics and combat. Previous, classic battles are also replayed for commanders, both as they originally played out and to explore alternative responses. War colleges pick apart the battles of Hannibal and Napoleon in much the same way as historical gaming circles, and commanders replay the tank tactics used in the 73 Easting battle, observing with displays similar to those of today's combat-oriented games.

Chapter 3

Combat simulator theories and 3D graphics engine

3.1Physics in 3D:

As long as we only have single floating point values for position and velocity our physics simulation is limited to motion in a single dimension, and a point moving from side to side on the screen is not our goal.

We want our object to be able to move in three dimensions: left and right, forward and back, up and down. If we apply the equations of motion to each dimension separately, we can integrate each dimension in turn to find the motion of the object in three dimensions.

Or we could just use vectors.

Vectors are a mathematical type representing an array of numbers. A three dimensional vector has three components x, y and z. Each component corresponds to a dimension. In this article x is left and right, y is up and down, and z is forward and back.

We can implement vectors using a struct as follows:

```
struct Vector
   {
      float x,y,z;
   };
```

Addition of two vectors is defined as adding each component together. Multiplying a vector by a floating point number is the same as just multiplying each component. We can add overloaded operators to the vector struct so that it is possible to perform these operations in code as if vectors are a native type:

```
struct Vector
   {
      float x,y,z;

      Vector operator + ( const Vector &other )
      {
         Vector result;
         result.x = x + other.x;
         result.y = y + other.y;
         result.z = z + other.z;
         return result;
      }

      Vector operator*( float scalar )
      {
         Vector result;
         result.x = x * scalar;
         result.y = y * scalar;
         result.z = z * scalar;
         return result;
      }
   };
```

Now instead of maintaining completely separate equations of motion and integrating separately for x, y and z, we convert our position, velocity, acceleration and force to vector quantities, then integrate the vectors directly using the equations of motion.

$$\mathbf{F} = m\mathbf{a}$$
$$d\mathbf{v}/dt = \mathbf{a}$$
$$d\mathbf{x}/dt = \mathbf{v}$$

Now that we have the equations of motion in vector form, the question of integrating them comes in. The answer is exactly the same as we integrated single values. This is because we have already added overloaded operators for adding two vectors together, and multiplying a vector by a scalar and this is all we need to be able to drop in vectors in place of floats and have everything just work.

For example, here is a simple Euler integration for vector position from velocity:

```
position = position + velocity*dt;
```

Notice how the overloaded operators make it look exactly the same as an Euler integration for a single value. But what is it really doing? Let's take a look at how we would implement vector integration without the overloaded operators:

```
position.x = position.x + velocity.x * dt;
position.y = position.y + velocity.y * dt;
position.z = position.z + velocity.z * dt;
```

As we can see, it's exactly the same as if we integrated each component of the vector separately.This could be the first theoretical points to understand 3D world in virtual space for building the type of the simulator we are trying to build.

3.2 First person shooters:

First person shooter physics are usually very simple. The world is static and players are limited to running around and jumping and shooting. Because of cheating, first person shooters typically operate on a client-server model where the server is authoritative over physics. This means that the true physics simulation runs on the server and the clients display an approximation of the server physics to the player.

Fig 3.1: The view of a First Person Shooter video game

The problem then is how to allow each client to control his own character while displaying a reasonable approximation of the motion of the other players.

In order to do this elegantly and simply, we structure the physics simulation as follows:

1. Character physics are completely driven from input data
2. Physics state is known and can be fully encapsulated in a state structure
3. The physics simulation is reasonably deterministic given the same initial state and inputs

To do this we need to gather all the user input that drives the physics simulation into a single structure and the state representing each player character into another.

3.3 Client Server Network Model:

3.3.1 Peer-to-Peer Lockstep:

In the beginning games were networked peer-to-peer, between computers exchanging information with each other in a fully connected mesh topology. We can still see this model alive today in RTS games, and interestingly for some reason, perhaps because it was the first way – it's still how most people think that game networking works.

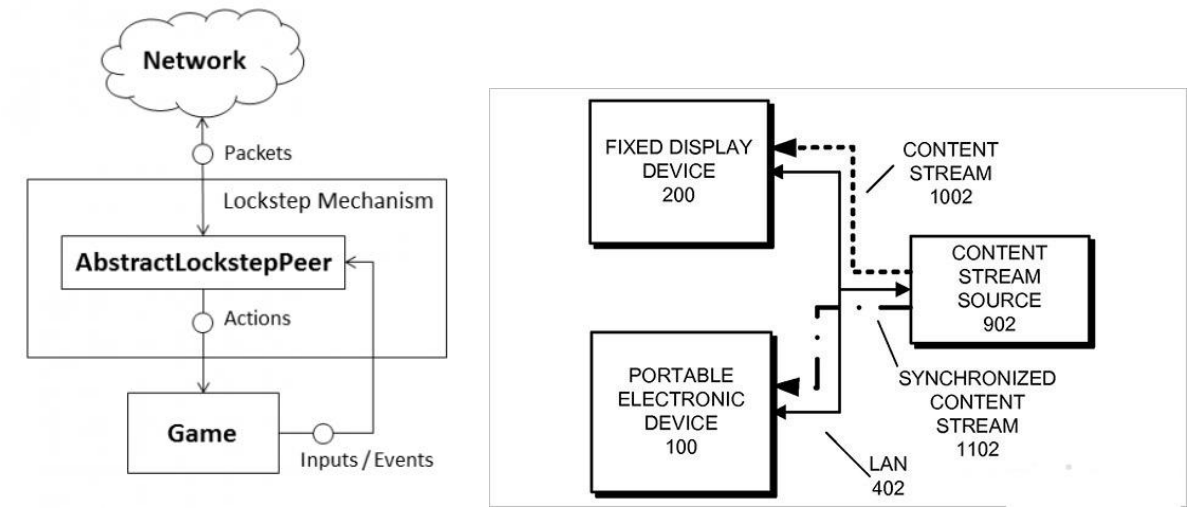

Fig 3.2: The Flow Diagram of P2P Lockstep model

The basic idea is to abstract the game into a series of turns and a set of command messages when processed at the beginning of each turn direct the evolution of the game state. For example: move unit, attack unit, construct building. All that is needed to network this is to run exactly the same set of commands and turns on each player's machine starting from a common initial state.

It seems simple and elegant, but unfortunately there are several limitations.

First, it's exceptionally difficult to ensure that a game is completely deterministic; that each turn plays out identically on each machine. For example, one unit could take slightly a different path on two machines, arriving sooner to a battle and saving the day on one machine, while arriving later on the end.

The next limitation is that in order to ensure that the game plays out identically on all machines it is necessary to wait until all players' commands for that turn are received beforesimulating that turn. This means that each player in the game has latency equal to the most lagged player. RTS games typically hide this by providing audio feedback immediately and/or playing cosmetic animation, but ultimately any truly game affecting action may occur only after this delay has passed.

The final limitation occurs because of the way the game synchronizes by sending just the command messages which change the state. In order for this to work it is necessary for all players to start from the same initial state. Typically this means that each player must join up in a lobby before commencing play, although it is technically possible to support late join, this is not common due to the difficulty of capturing and transmitting a completely deterministic starting point in the middle of a live game.

3.3.2 Client/Server:

In a pure client/server model we run no game code locally, instead sending our inputs such as key presses, mouse movement, clicks to the server. In response the server updates the state of our character in the world and replies with a packet containing the state of our character and other players near us. All the client has to do is interpolate between these updates to provide the illusion of smooth movement and we have a networked client/server simulation.

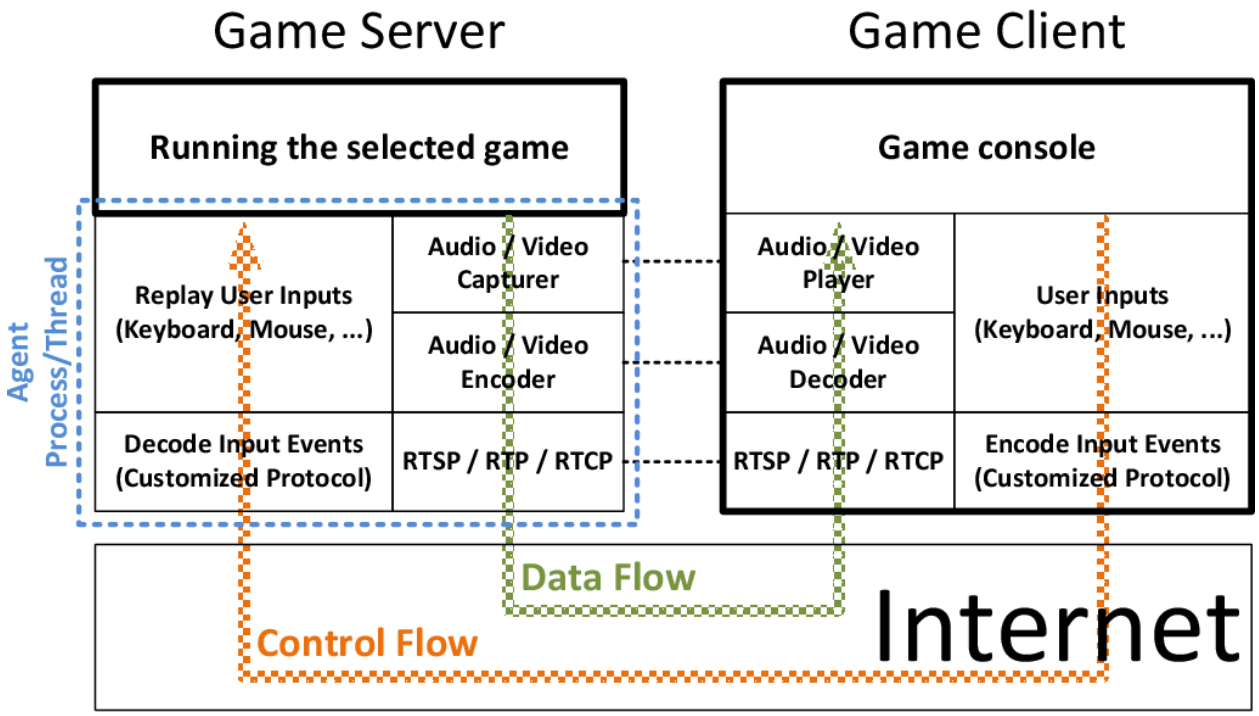

Fig 3.3: The basic architecture model of a Client-Server based gaming

The quality of the game experience now depended on the connection between the client and the server instead of the most lagged peer in the game. It also became possible for players to come and go in the middle of the game, and the number of players increased as client/server reduced the bandwidth required on average per-player.

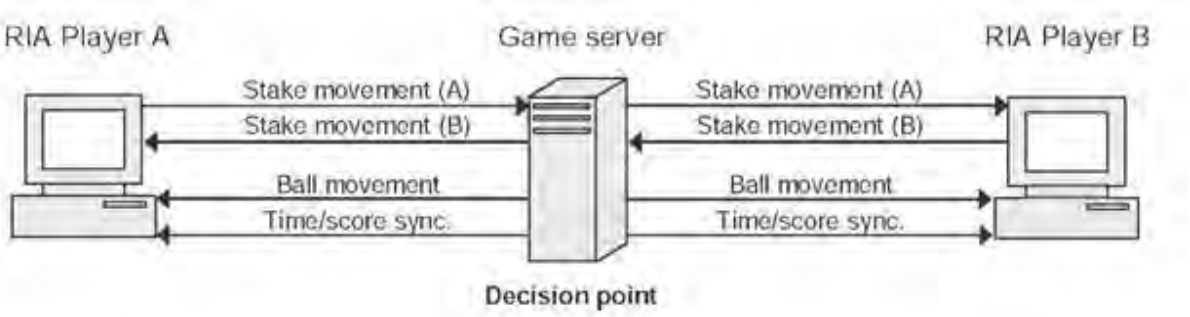

Fig 3.4: The mechanics of a Client-Server based model with two players

3.4 Unity 3D:

With an emphasis on portability, the engine targets the following APIs: Direct3D on Windows, OpenGL on Mac and Windows. Unity allows specification of texture compression and resolution settings for each platform the game engine supports, and provides support for mapping, reflection, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects. Unity's graphics engine's platform diversity can provide a shader with multiple variants and a declarative fallback specification, allowing Unity to detect the best variant for the current video hardware; and if none are compatible, fall back to an alternative shader that may sacrifice features for performance.

3.5 Photon Networking Framework:

Low latency is an essential requirement in real-time multiplayer games. For this reason Photon Cloud is hosted in all major world regions to provide our players with a minimum latency.Photon is a real-time multiplayer game development framework which has server and cloud services. Games that depend on low latency, like FPS or RTS game types, connect the players to the nearest region. Games that are able to handle higher latency, like turn based game types can connect all players to the same region.

For our projects, we prefer to implement the networking functionality early during development. This means for each new feature, we make sure it also works over the network. In the end this saves us a lot of time, because implementing it at a later stage results in changing a lot of code.

When we start the simulator, we want to connect to the Photon network. This will enable us to host and join rooms of our game based on the AppID. For the function call PhotonNetwork.ConnectUsingSettings() we can add the version number of the game as a string.

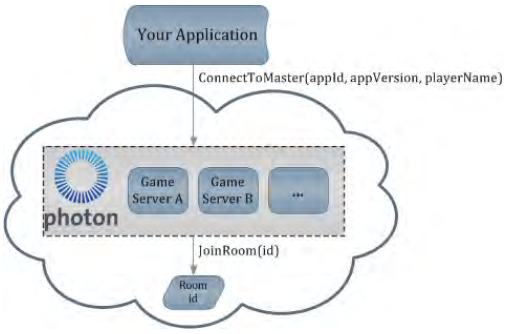

Fig 3.5: Photon Server Connectivity Authentication

After the connection has been established, we need to connect to a room. A room can be created with the function call *PhotonNetwork.CreateRoom()*, where we have to define a unique room

name. In the example below a unique ID is added to the name. Other parameters can be used to hide or close the room or set the maximum number of players.

3.6 Lighting Theory for 3D Simulators:

Lighting is traditionally one of the slower or "expensive" things to calculate and render in a game engine. Considering the science of visible light: countless photons at different wavelengths bouncing around at unimaginable speeds that somehow enter our eye. To do any of this at a reasonable frame rate, game engines must strategically simplify light calculations in specific ways, and then hope players don't notice the inconsistencies. So how can we in 3D game engines generally do lighting?

The simplest form of lighting is **"ambient light"**, a constant default light level to apply to every model in the world, even the parts in shadow. This isn't realistic at all, so the ambient light usually serves more as a "fill light", especially for outdoor spaces, where it is often a dark navy color to make sure the shadows aren't terminating into pitch-black.

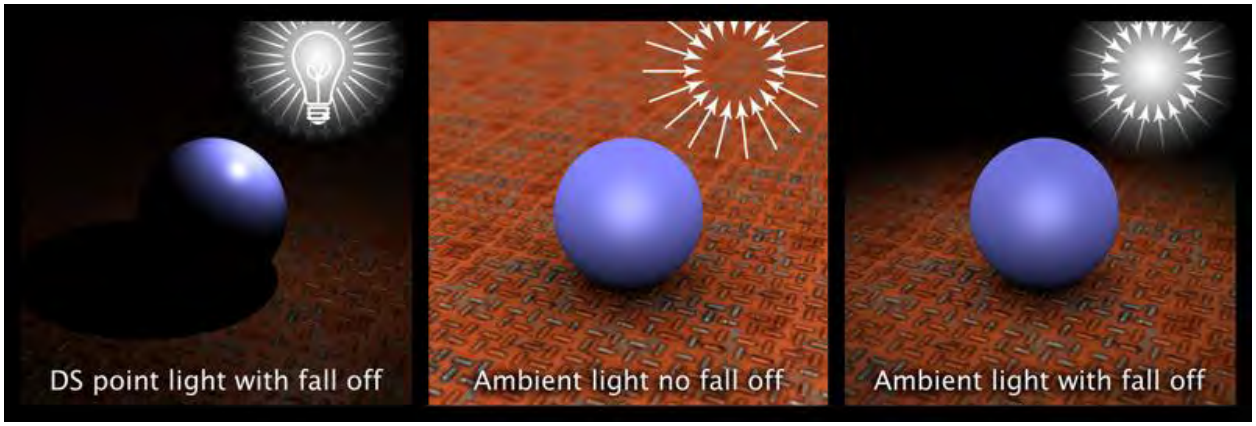

Fig 3.6: Ambient Lighting Effect

A **"directional light"** is for sunlight or any other global light source, casting a constant light from a given rotation. It affects the entire world all at once, so it can be placed anywhere, even inside a wall; only its direction matters. Again, a directional light will shine on every surface facing toward it, and distance from this sun / moon does not matter. Because most scenes and game worlds will only have a single directional light, they usually function more like "key lights."

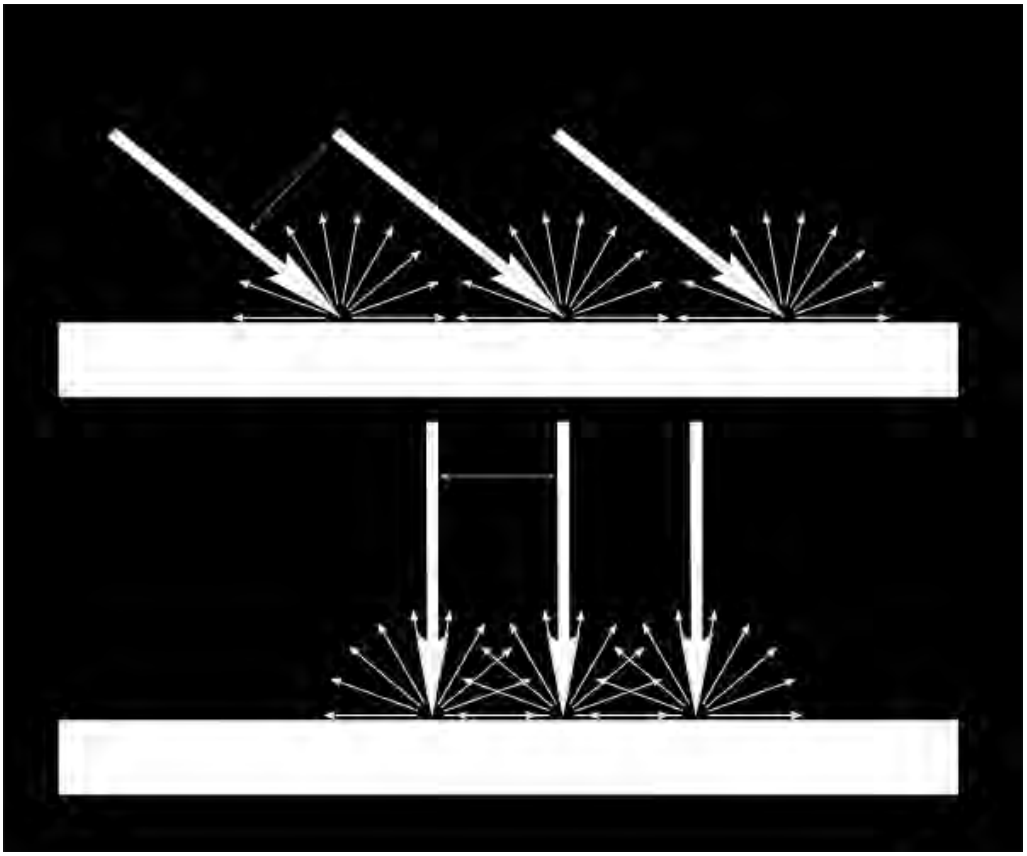

Fig 3.7: Diffuse Angle

A **"spotlight"** casts an X degrees-wide cone of light at Y intensity. If we were making a game with lots of stage lighting, recessed ceiling lighting, or street lamps, then we would probably rely heavily on spotlights. The most common use of a spotlight is an elevated spotlight pointing

down, de-emphasizing the ceiling and highlighting the floor instead. In this way, spotlights are great for implying specific directions or emphasizing specific places, whatever they're shining on.

Lastly, a **"point light"** is an invisible omnidirectional ball of light that casts light in all directions at X intensity. We would commonly use this type for things like table lamps, cage lights, or chandeliers. Point lights are really good at drawing attention to a given light source, because they generally have to be near whatever they're illuminating and all nearby shadow-casting objects will practically be pointing to it.

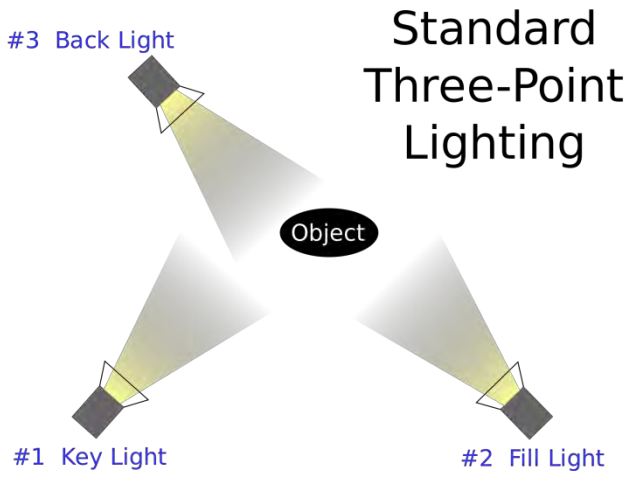

Fig 3.8: The effect and usability of point lighting

3.7 Camera and Perspective:

A camera is a simple entity that resides in world space. It provides the origin of the view frustum, which is a pyramid encompassing what is visible in the world (this is usually where we are in the world). The camera also specifies what we are looking at, and where up and down is. All information in the camera are defined in world space. It is really handy to use a 3x3 matrix

and a translation vector to represent the camera information. The first vector is where "right" is, the second is "up" and the third is the direction we are looking at.

```
|UxVxNx|
|UyVyNy|
|UzVzNz|
```

Where U is the "right" vector, V the "up" vector and N the direction we are looking. The Translation vector is simply the location of the camera.
For example, if we are at (3,10,5), right is along vector (1,0,0), up along vector (0,0,1) and we are looking at something at point (3,15,5). The matrix representation would is:

```
|100|
|001|
|010|
```

We can think of a viewport as a kind of window. The viewport delimits what is visible to the camera. It defines the sides of the view frustum or "viewing pyramid". The viewport defines the distance from the camera to the projection plane. The viewport also defines a rectangle on the projection plane, and anything outside of this rectangle is not visible. In effect, this rectangle defines the field of view, or FOV. The FOV is usually represented with an angle, which is the angle formed by the sides of the rectangle with the camera location.

Chapter 4

Implementation:

4.1 Character Controller:

It defines the characteristics of each individual player and how they respond to each and every input from the user. It also contains how the characters interact with the environment within the simulation itself.

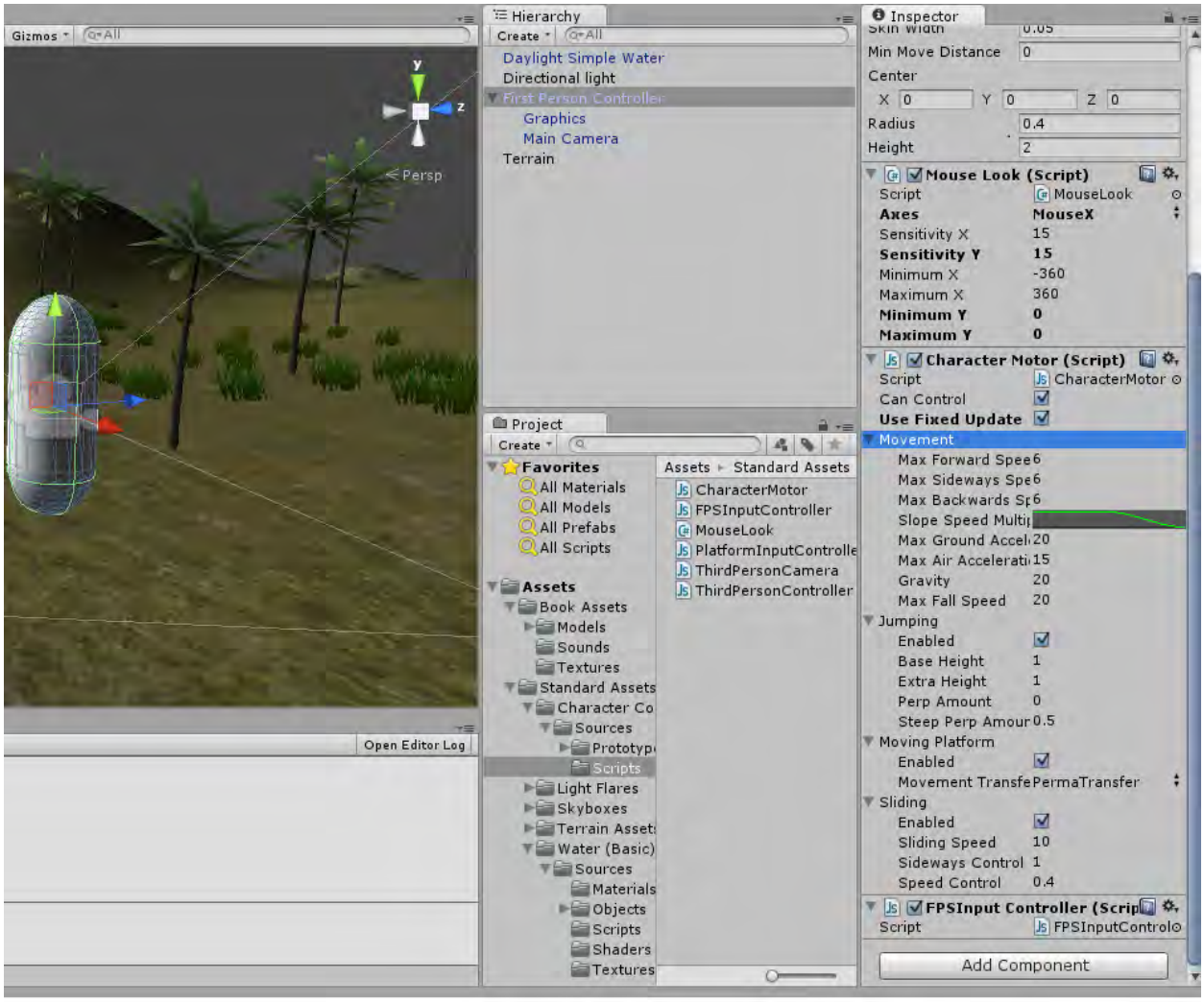

Fig 4.1: FPS Character Controller

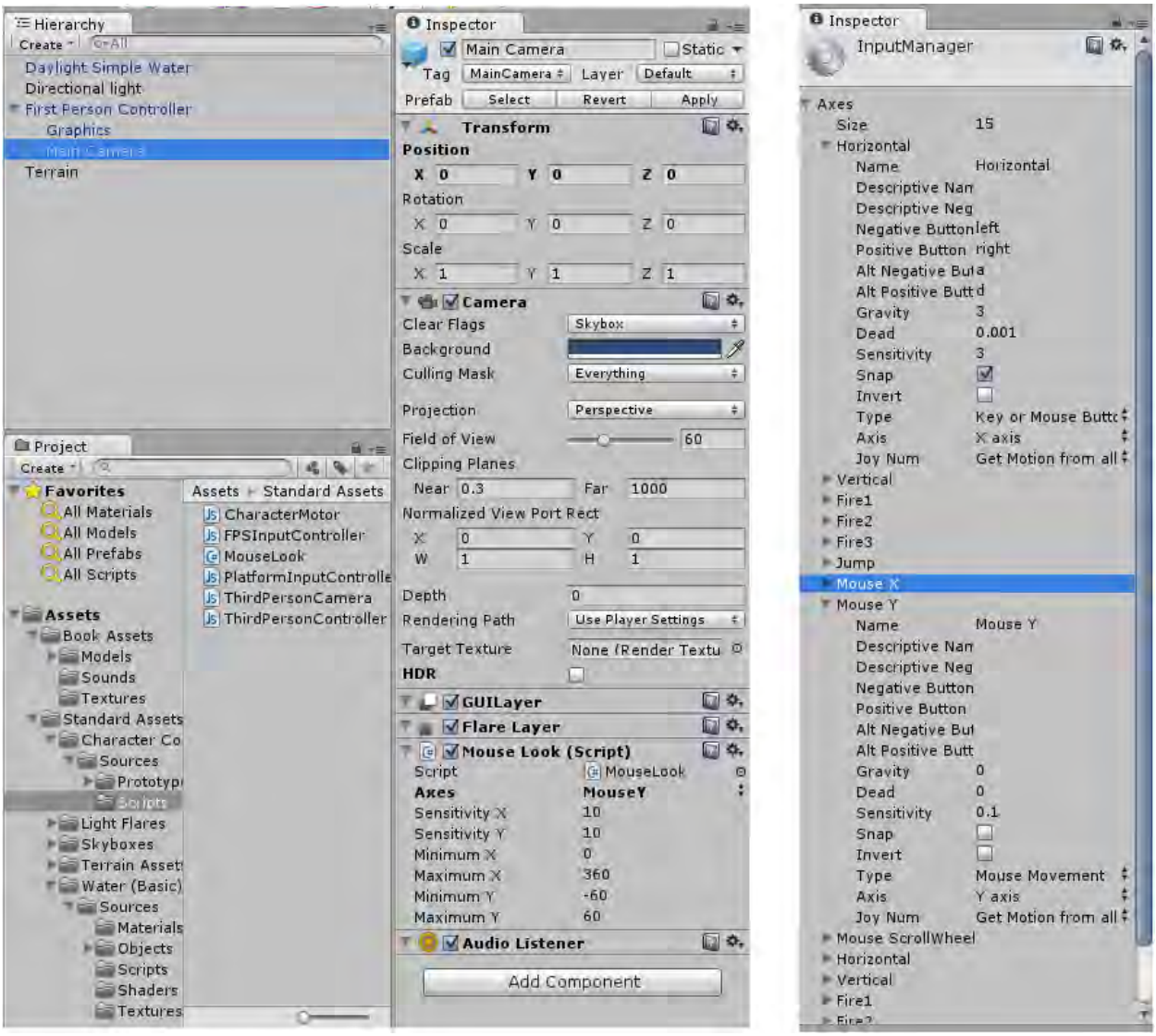

Fig 4.2: FPS Character Controller (Continued)

4.2 Directional Controller:

Although Unity3D has its own mouse look controller, we built our like the character controller. It acts quite similar to the one Unity3D provides but has smoother movements and physics properties optimized for our simulator. It provides constant frames per second thus making the simulator more realistic and efficient.

4.3 Object Animation:

The objects in the simulator which animates such as the players and guns are animated using Blender. The animation is associated with each respective objects and different method can be individually instantiate while implementing in Unity3D.

Fig 4.3: Character Animation

4.4 Weapon Mechanics:

Implementing the weapon firing, switching and hit points mechanics is the major part of our build which turns few animating objects into real time simulator. The mechanics has two different C# scripts. The “wepScript” is the main script which contains the default weapon mechanics.The second script “wepSwitcher” is implanted if the situation requires assigning multiple weapons to a player. So the player can eventually switch between those weapons.

Fig 4.4: Weapon Animation

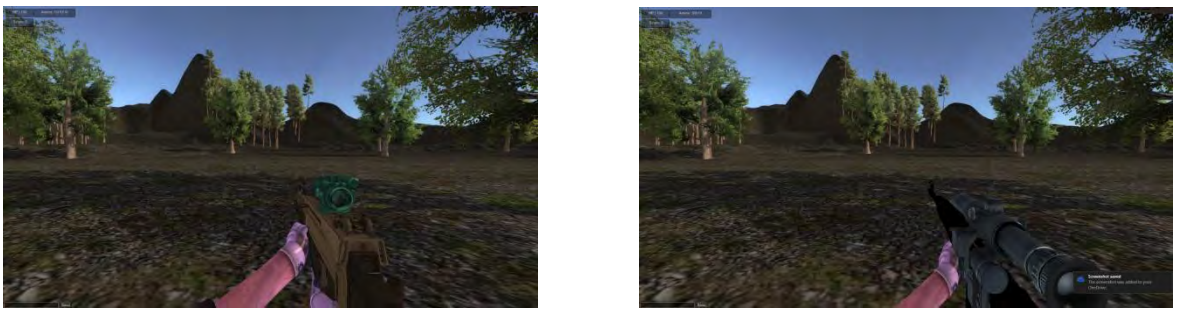

Fig 4.5: Different Weapons

4.5. Photon Unity Network Manager:

We implemented PUN (Photon Unity Network Manager) for creating and hosting lobbies for other players to join. The “roomMan” in implanted as an object. In our case the PUN is associated with the main camera as unity can only associate scripts with an object. It spawns player every time a player joins or creates a lobby and updates the database of the main server throughout the simulation.

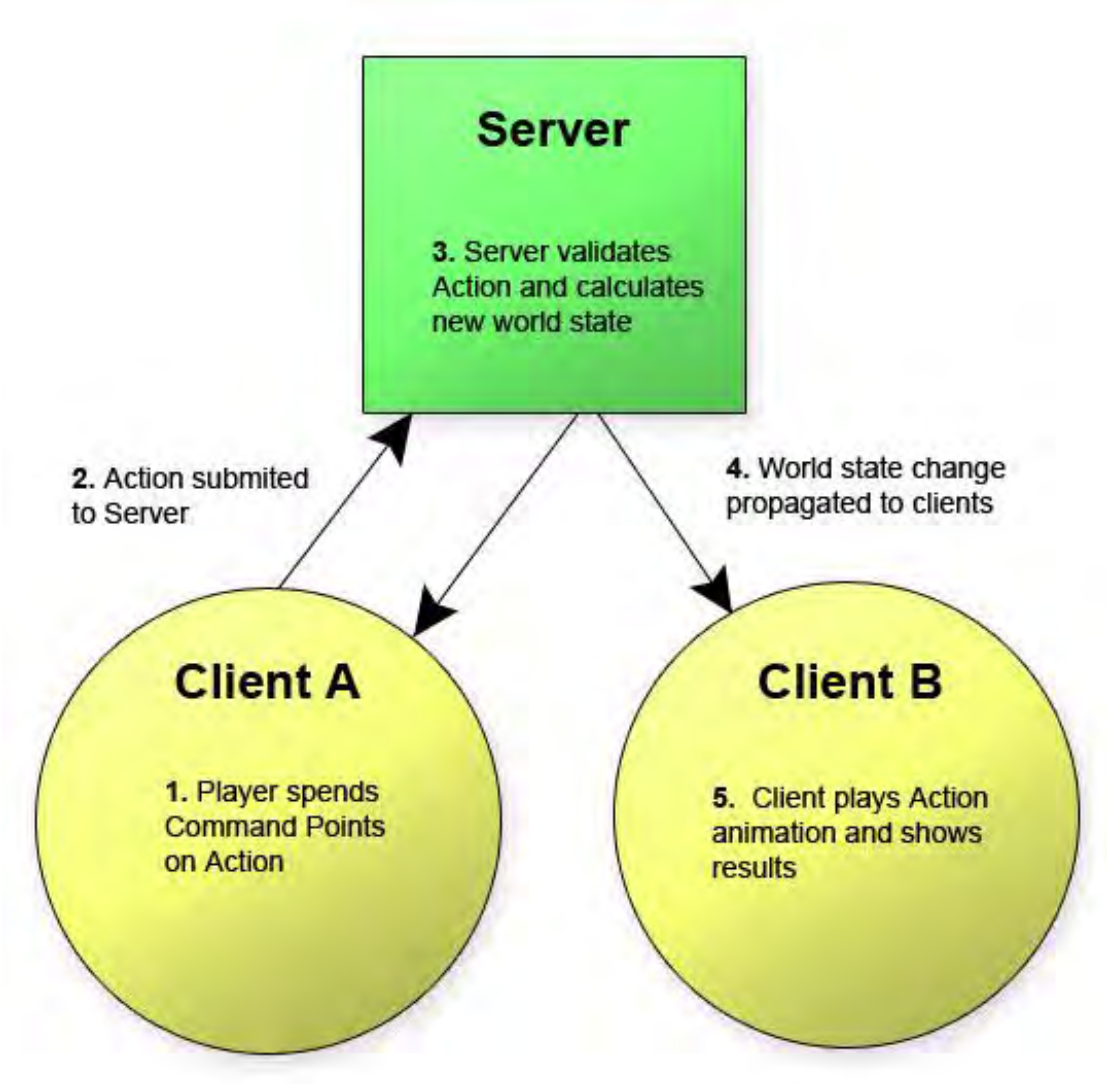

Fig 4.6: Implementation of the Photon Client-Server system

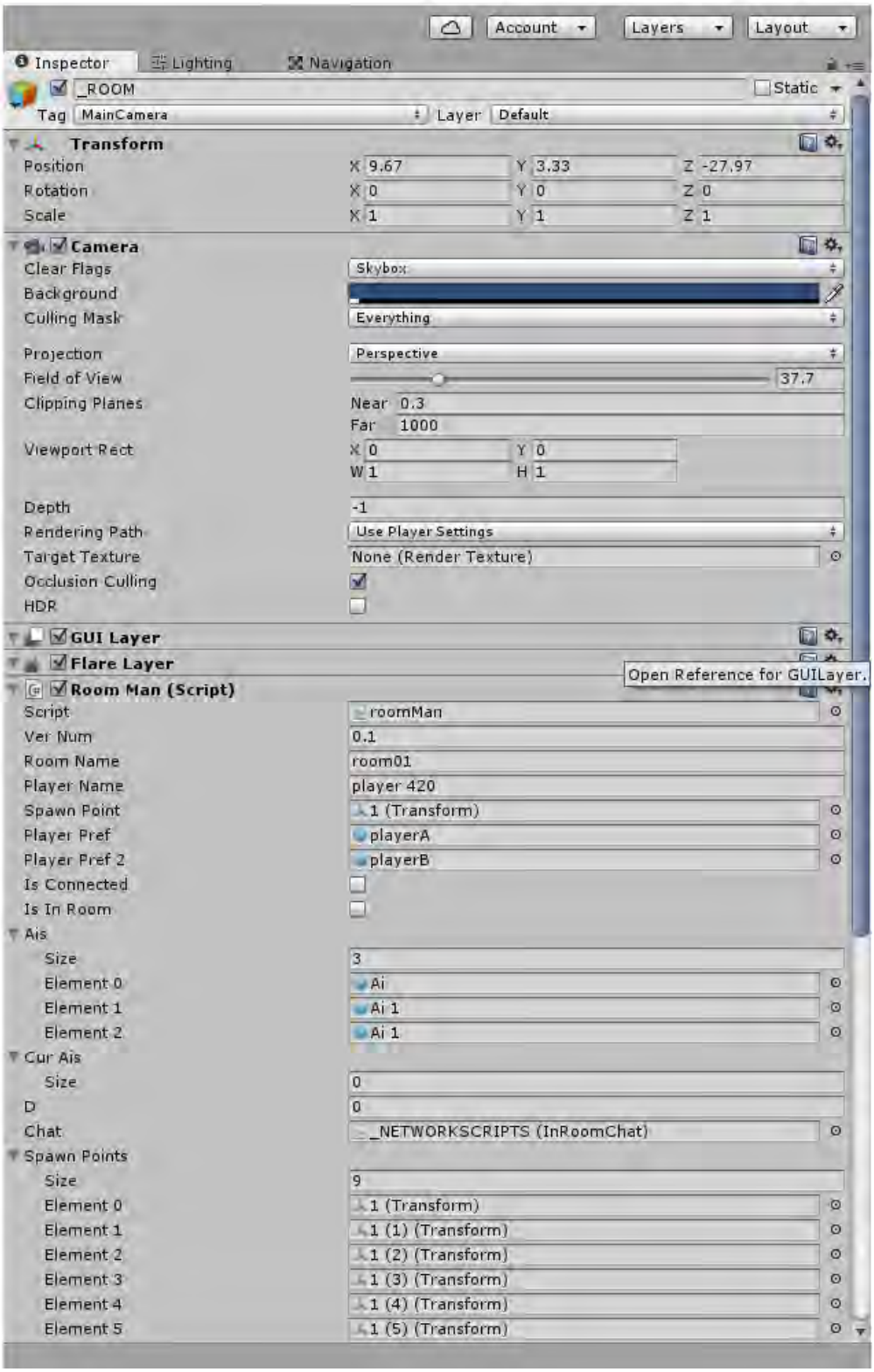

Fig 4.7: PUN Room Manager

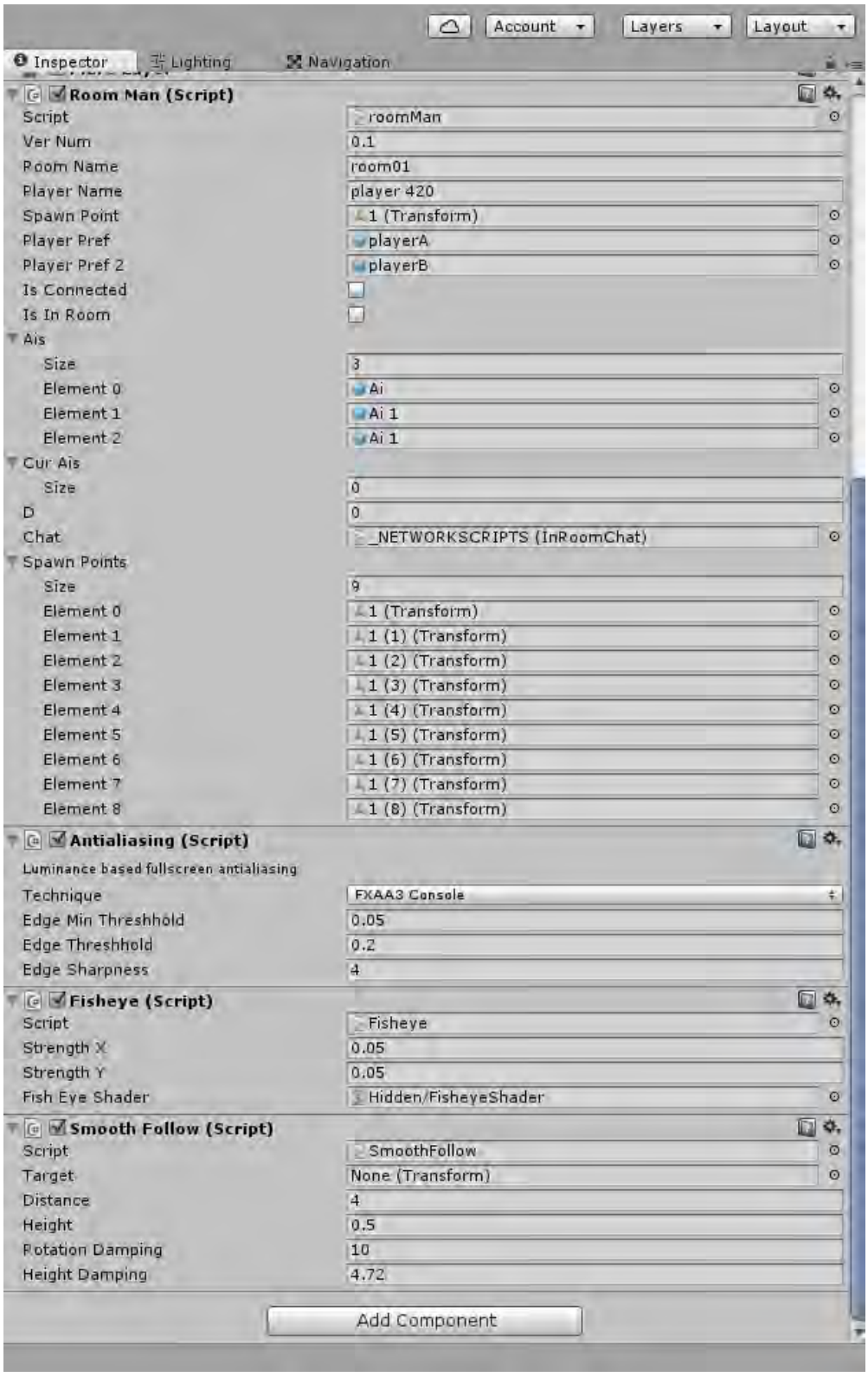

Fig 4.8: PUN Room Manager (Continued)

5. Discussion and Future Work:

As we discussed before that we build this project as a prototype to demonstrate the possibility of next of warfare. Most of the developed countries have their own multimillion dollar war simulators for their militaries and that is where 3rd world countries like us fall behind. Our project has many limitations as it is mainly build on open source software and free game engine. But the completion of our project signifies that with proper logistics and budget, this project can go a long way, even becoming the heart of modern warfare in our national defense system.

For future work, we would like to establish a dedicated server rather than using a default photon server and also aim towards building a more realistic graphical model and terrain to enhance the simulation experience

Bibliography

1. Brown, D. (1999). Civil War Simulation Games. *The Review: A Journal of Undergraduate Student Researc*.
2. de, j. v. (2014, May 8). *how to create an online multiplayer game with photon unity networking.* Retrieved from www.paladinstudios.com: http://www.paladinstudios.com/2014/05/08/how-to-create-an-online-multiplayer-game-with-photon-unity-networking/
3. fielder, G. (n.d.). *game-physics*. Retrieved from gafferongames.com: http://gafferongames.com/game-physics/
4. Grammatico, J. (1990). *USA Patent No. US4902017 A.*
5. H.Addink, D. (2000). *USA Patent No. US6042477 A.*
6. *lighting design theory for 3D games.* (2015, january 29). Retrieved from www.blog.radiator.debacle.us: http://www.blog.radiator.debacle.us/2015/01/lighting-design-theory-for-3d-games.html
7. Macedonia, M. (n.d.). Games,Simulation and the Military Education Dilemma.
8. *Military Spending in the United States*. (n.d.). Retrieved from National priorities project: https://www.nationalpriorities.org/campaigns/military-spending-united-states/
9. Ngok, L. (n.d.). A simple bound on the game server computing power .
10. Stumfol, S. (2014). Implementation of Character Controls and Combat system in the Action Adventures "Scout COD" realized with unity 3D.

Appendix

```
using UnityEngine;
using System.Collections;

public class RigidbodyFPSWalker : MonoBehaviour {


        public float speed = 10.0f;
        public float gravity = 10.0f;
        public float maxVelocityChange = 10.0f;
        public bool canJump = true;
        public float jumpHeight = 2.0f;
        private bool grounded = false;
        public PhotonView name;

        public int kills = 0;
        public int deaths = 0;


        public Texture blood;
        public int bloodTimer;

        public string theKiller = "";
        public string killerWep = "";

        public GameObject me;
        public GameObject graphics;
        public GameObject ragDoll;
        public GameObject[] bots;

        public int health = 100;
```

```
public GameObject fpsCam;

public bool isPause = false;

public AudioSource sound1;
public AudioClip soundClip;

public GameObject[] activePlayers;



void Awake () {
        GetComponent<Rigidbody>().freezeRotation = true;
        GetComponent<Rigidbody>().useGravity = false;
        name.RPC ("updateName", PhotonTargets.AllBuffered, PhotonNetwork.playerName, health);
        gameObject.name = PhotonNetwork.playerName;

}

void FixedUpdate () {
        kills = PlayerPrefs.GetInt ("kills");
        deaths = PlayerPrefs.GetInt ("deaths");




        if (grounded) {
                // Calculate how fast we should be moving
                Vector3 targetVelocity = new Vector3(Input.GetAxis("Horizontal"), 0,
Input.GetAxis("Vertical"));
                targetVelocity = transform.TransformDirection(targetVelocity);
                targetVelocity *= speed;

                // Apply a force that attempts to reach our target velocity
                Vector3 velocity = GetComponent<Rigidbody>().velocity;
```

```
                Vector3 velocityChange = (targetVelocity - velocity);
                velocityChange.x = Mathf.Clamp(velocityChange.x, -maxVelocityChange,
maxVelocityChange);
                velocityChange.z = Mathf.Clamp(velocityChange.z, -maxVelocityChange,
maxVelocityChange);
                velocityChange.y = 0;
                GetComponent<Rigidbody>().AddForce(velocityChange, ForceMode.VelocityChange);


                // Jump
                if (canJump && Input.GetButton("Jump")) {
                        GetComponent<Rigidbody>().velocity = new Vector3(velocity.x,
CalculateJumpVerticalSpeed(), velocity.z);
                }
        }


        bloodTimer += -1;


        if (Input.GetKeyDown (KeyCode.Escape)) {
                isPause = true;
        }
        if(Input.GetKeyUp (KeyCode.Escape)){
                isPause = false;
        }


        if (isPause) {
                Cursor.lockState = CursorLockMode.None;
                Cursor.visible = true;
        } else {
                Cursor.lockState = CursorLockMode.Locked;
                Cursor.visible = false;
        }


        // We apply gravity manually for more tuning control
        GetComponent<Rigidbody>().AddForce(new Vector3 (0, -gravity *
GetComponent<Rigidbody>().mass, 0));
```

```
		grounded = false;

		if (health <= 0) {
			GetComponent<PhotonView>().RPC ("DIE", PhotonTargets.AllBuffered, null);
		}

		bots = GameObject.FindGameObjectsWithTag("Ai");
	}

	void OnCollisionStay () {
		grounded = true;
	}

	float CalculateJumpVerticalSpeed () {
		// From the jump height and gravity we deduce the upwards speed
		// for the character to reach at the apex.
		return Mathf.Sqrt(2 * jumpHeight * gravity);
	}



	void OnGUI(){
		GUI.Box (new Rect (10, 10, 100, 30), "HP | " + health);

		if (bloodTimer >= 1) {
			GUI.DrawTexture (new Rect(0,0,Screen.width, Screen.height), blood,
ScaleMode.StretchToFill);
		}



		if (isPause) {
```

```
                GUILayout.BeginArea (new Rect(Screen.width/2 - 250, Screen.height/2 - 250, 500,500));
                GUILayout.Box ("ActiveBots: " + "x" + bots.Length);
                GUILayout.Box ("Score:" + "\nKills: " + kills + " / " + "Deaths: " + deaths);
                GUILayout.Box ("Players:    Name        K/D   ");
                foreach(PhotonPlayer pl in PhotonNetwork.playerList){
                        GUILayout.Box (pl.name + " | " + pl.GetScore ());
                }
                if(GUILayout.Button ("Disconnect")){
                        PhotonNetwork.Disconnect ();
                        Application.LoadLevel (0);
                }
                GUILayout.EndArea ();
        }
    }

    void OnCollisionEnter(Collision col){
        if (col.transform.tag == "Bullet") {
                Debug.Log ("Hit");
                int dmg = col.transform.GetComponent<bulScript>().dmg;
                string aiName = col.transform.GetComponent<bulScript>().name;
                GetComponent<PhotonView>().RPC ("applyDamage", PhotonTargets.All, dmg, aiName,
"M4");
        }
    }

    public void getClip (AudioClip soundToRecieve){
        Debug.Log (soundToRecieve.name);
        soundClip = soundToRecieve;
    }



    [PunRPC]
    public void applyDamage(int dmg, string killerName, string wepName){
```

```
            health = health - dmg;

            //GameObject.Find (killerName).GetComponent<PhotonView>().RPC ("addKill",
PhotonTargets.AllBuffered);

            name.RPC ("updateName", PhotonTargets.AllBuffered, PhotonNetwork.playerName, health);

            theKiller = killerName;

            killerWep = wepName;

            Debug.Log ("hit!" + health);

            bloodTimer = 20;

        }


        [PunRPC]

        public void DIE(){

            if (GetComponent<PhotonView> ().isMine) {

                PhotonNetwork.Destroy (me);

                Destroy (me);

                GameObject rDoll = PhotonNetwork.Instantiate (ragDoll.name, transform.position,
transform.rotation, 0);

                Destroy (rDoll, 3);

                GameObject.Find ("_ROOM").GetComponent<roomMan> ().OnJoinedRoom ();

                GameObject.Find ("_NETWORKSCRIPTS").GetComponent<PhotonView>().RPC
("addFeed", PhotonTargets.All, theKiller + " [" + killerWep + "] " + PhotonNetwork.playerName);


                Debug.Log ("die");

                PlayerPrefs.SetInt ("deaths", deaths + 1);


                GameObject killer = GameObject.Find (theKiller);

                killer.GetComponent<PhotonView>().RPC ("exitTrig", PhotonTargets.AllBuffered, null);

                killer.GetComponent<PhotonView>().RPC ("addKill", PhotonTargets.AllBuffered, null);



            }



        }
```

```
        [PunRPC]
        public void addKill(){
                PlayerPrefs.SetInt ("kills", kills + 1);
                Debug.Log ("added kill! " + kills);
        }


        public AudioClip[] sounds;


        [PunRPC]
        public void playSound(){
                Debug.Log ("Sound: " + soundClip.name);
                sound1.PlayOneShot (soundClip);
        }
}


using UnityEngine;
using System.Collections;

public class animManager : MonoBehaviour {

        public Animation am;
        public AnimationClip walk;
        public AnimationClip idle;

        public bool isTP = false;

        public PhotonView graphicsPV;
        public Animation graphicsAM;
        public AnimationClip tpRun;
        public AnimationClip tpIdle;
        public AnimationClip tpShoot;
```

```
public AnimationClip tpRunShoot;
public AnimationClip tpReload;
public wepScript ws;
public Rigidbody rb;

void Update(){

	if (rb.velocity.magnitude >= 0.1) {
		if(!isTP){
			playAnim (walk.name);
		} else {
			graphicsPV.RPC ("playAnimPV", PhotonTargets.All, tpRun.name);
		}
	} else {
		if(!graphicsAM.IsPlaying(tpReload.name)){
			if(!isTP){
				playAnim (idle.name);
			} else {
				graphicsPV.RPC ("playAnimPV", PhotonTargets.All, tpIdle.name);
			}
		}
	}

	if (Input.GetKeyDown (KeyCode.R) && ws.clipCount >= 1) {
		if(isTP && graphicsPV.isMine){
			graphicsPV.RPC ("playAnimPV", PhotonTargets.All, tpReload.name);
		}
	}

	if (Input.GetMouseButtonDown (0) && ws.ammo >= 1) {
		if(isTP){
			graphicsPV.RPC ("playAnimPV", PhotonTargets.All, tpShoot.name);
		}
	}
```

```
	}


	public void playAnim(string animName){
		if (!isTP) {
			am.CrossFade (animName);
		}
	}

	public void stopAnim(){
		am.Stop ();
	}

	public void reload(){
		graphicsPV.RPC ("playAnimPV", PhotonTargets.All, tpReload.name);
	}

	[PunRPC]
	public void playAnimPV(string animName){
		if (isTP) {
			graphicsAM.CrossFade (animName);
		}
	}
}

using UnityEngine;
using System.Collections;

public class bulScript : MonoBehaviour {

	public int dmg;
	public string name;
```

```
	void Awake(){
		gameObject.name = "bullet";
	}

	void OnCollisionEnter(){
		Destroy (gameObject);
	}

}
```

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class playerName : MonoBehaviour {

	public Text nameTag;



	[PunRPC]
	public void updateName(string name, int health){
		nameTag.text = name + " / " + health;
		Debug.Log (nameTag.text);
	}

}
```

```
using UnityEngine;
using System.Collections;

public class wepScript : MonoBehaviour {

	public Camera fpsCam;
	public GameObject hitPar;
	public int damage = 30;
	public int maxDamage = 60;
	public int range = 10000;
	public int ammo = 10;
	public int clipSize = 10;
	public int clipCount = 5;
	public float recoilPower = 30;
	public Animation am;
	public AnimationClip shoot;
	public AnimationClip reloadA;

	public Texture hitMarker;
	public int hitCD = 0;

	public AudioClip shootSound;
	public AudioClip reloadSound;

	public animManager amM;

	public string  weaponName = "";

	public GameObject[] objectsToDisable;
	public bool canAim = false;
	public float aimFOV = 20;
	public float regFOV = 60;
	public float regOffset = 0;
	public float aimOffset = -75;
```

```
public PhotonView pv;

public Texture scope;
public bool isAimed = false;

void Awake(){

}

void Update(){
	if (Input.GetMouseButton (0)) {
		fireShot ();
	}

	if (Input.GetMouseButtonDown (1)) {
		aim (true);
	}

	if (Input.GetMouseButtonUp (1)) {
		aim (false);
	}

	if (Input.GetKeyDown (KeyCode.R)) {
		reload();
	}

	hitCD += -1;
}

public void fireShot(){
	if (!am.IsPlaying (reloadA.name) && ammo >= 1) {
		if(!am.IsPlaying (shoot.name)){
			am.CrossFade (shoot.name);
```

```
                ammo = ammo - 1;

                pv.transform.GetComponent<RigidbodyFPSWalker>().getClip(shootSound);
                Debug.Log ("playing Sound!");
                pv.RPC ("playSound", PhotonTargets.AllBuffered, null);

                fpsCam.transform.Rotate (Vector3.right, -recoilPower * Time.deltaTime);

                RaycastHit hit;
                Ray ray = fpsCam.ScreenPointToRay (new Vector3 (Screen.width / 2, Screen.height
/ 2 + regOffset, 0));

                if (Physics.Raycast (ray, out hit, range)) {
                        if (hit.transform.tag == "Player") {
                                hit.transform.GetComponent<PhotonView> ().RPC
("applyDamage", PhotonTargets.AllBuffered, Random.Range (damage, maxDamage), PhotonNetwork.playerName,
weaponName);
                                PhotonNetwork.player.AddScore(1);
                                Debug.Log (PhotonNetwork.player.GetScore());
                                hitCD = 20;
                        }

                        if (hit.transform.tag == "Ai") {
                                hit.transform.GetComponent<PhotonView> ().RPC
("AiDamage", PhotonTargets.AllBuffered, Random.Range (damage, maxDamage), PhotonNetwork.playerName,
weaponName);
                                PhotonNetwork.player.AddScore(1);
                                Debug.Log (PhotonNetwork.player.GetScore());
                                hitCD = 20;
                        }

                        GameObject particleClone;
                        particleClone = PhotonNetwork.Instantiate (hitPar.name, hit.point,
Quaternion.LookRotation (hit.normal), 0) as GameObject;
                        Destroy (particleClone, 2);
```

```
                    Debug.Log (hit.transform.name);

                }
            }

        }
    }

    public void reload(){
        if (clipCount >= 1) {
            am.CrossFade (reloadA.name);
            GetComponent<AudioSource>().PlayOneShot(reloadSound);
            ammo = clipSize;
            clipCount = clipCount - 1;
            amM.reload ();
        }

    }

    public void aim(bool isIn){
        if (canAim) {
            if (isIn) {
                fpsCam.fieldOfView = aimFOV;
                disable ();
                regOffset = aimOffset;
                isAimed = true;
            } else {
                fpsCam.fieldOfView = regFOV;
                enable ();
                regOffset = 0;
                isAimed = false;
            }
        }
    }
```

```
	public void disable(){
		foreach (GameObject part in objectsToDisable) {
			part.SetActive (false);
		}
	}
	public void enable(){
		foreach (GameObject part in objectsToDisable) {
			part.SetActive (true);
		}
	}



	void OnGUI(){
		GUI.Box (new Rect(110,10,150,30), "Ammo: " + ammo + "/" + clipSize + "/" + clipCount);
		GUI.Box (new Rect (10, 45, 100, 30), "Score: " + PhotonNetwork.player.GetScore());


		if (hitCD >= 0) {
			GUI.DrawTexture (new Rect(Screen.width/2 - 25, Screen.height / 2 -25, 50,50), hitMarker,
ScaleMode.StretchToFill);
		}



		if (isAimed) {
			GUI.DrawTexture (new Rect(0,0,Screen.width, Screen.height), scope,
ScaleMode.StretchToFill);
		}
	}
}
```

```
using UnityEngine;
using System.Collections;

public class wepSwitcher : MonoBehaviour {

	public GameObject[] weps;

	void Awake(){
		changeWep (0);
	}

	void Update(){

		if(Input.GetKeyDown (KeyCode.Alpha1)){
			changeWep (0);
		}

		if(Input.GetKeyDown (KeyCode.Alpha2)){
			changeWep (1);
		}

		if(Input.GetKeyDown (KeyCode.Alpha3)){
			changeWep (2);
		}



	}

	public void changeWep(int seq){
		disableAll ();
		weps [seq].SetActive (true);
	}

	public void disableAll(){
```

```
                foreach (GameObject wep in weps) {
                        wep.SetActive (false);
                }
        }

}
```

```
using UnityEngine;
using System.Collections;

[AddComponentMenu("Camera/Simple Smooth Mouse Look ")]
public class SimpleSmoothMouseLook : MonoBehaviour
{
        Vector2 _mouseAbsolute;
        Vector2 _smoothMouse;

        public Vector2 clampInDegrees = new Vector2(360, 180);
        public bool lockCursor;
        public Vector2 sensitivity = new Vector2(2, 2);
        public Vector2 smoothing = new Vector2(3, 3);
        public Vector2 targetDirection;
        public Vector2 targetCharacterDirection;

        // Assign this if there's a parent object controlling motion, such as a Character Controller.
        // Yaw rotation will affect this object instead of the camera if set.
        public GameObject characterBody;

        void Start()
        {
                // Set target direction to the camera's initial orientation.
                targetDirection = transform.localRotation.eulerAngles;
```

```
		// Set target direction for the character body to its inital state.
		if (characterBody) targetCharacterDirection = characterBody.transform.localRotation.eulerAngles;
	}

	void Update()
	{
		// Ensure the cursor is always locked when set
		Screen.lockCursor = lockCursor;

		// Allow the script to clamp based on a desired target value.
		var targetOrientation = Quaternion.Euler(targetDirection);
		var targetCharacterOrientation = Quaternion.Euler(targetCharacterDirection);

		// Get raw mouse input for a cleaner reading on more sensitive mice.
		var mouseDelta = new Vector2(Input.GetAxisRaw("Mouse X"), Input.GetAxisRaw("Mouse Y"));

		// Scale input against the sensitivity setting and multiply that against the smoothing value.
		mouseDelta = Vector2.Scale(mouseDelta, new Vector2(sensitivity.x * smoothing.x, sensitivity.y *
smoothing.y));

		// Interpolate mouse movement over time to apply smoothing delta.
		_smoothMouse.x = Mathf.Lerp(_smoothMouse.x, mouseDelta.x, 1f / smoothing.x);
		_smoothMouse.y = Mathf.Lerp(_smoothMouse.y, mouseDelta.y, 1f / smoothing.y);

		// Find the absolute mouse movement value from point zero.
		_mouseAbsolute += _smoothMouse;

		// Clamp and apply the local x value first, so as not to be affected by world transforms.
		if (clampInDegrees.x < 360)
			_mouseAbsolute.x = Mathf.Clamp(_mouseAbsolute.x, -clampInDegrees.x * 0.5f,
clampInDegrees.x * 0.5f);

		var xRotation = Quaternion.AngleAxis(-_mouseAbsolute.y, targetOrientation * Vector3.right);
		transform.localRotation = xRotation;
```

```
                // Then clamp and apply the global y value.
                if (clampInDegrees.y < 360)
                        _mouseAbsolute.y = Mathf.Clamp(_mouseAbsolute.y, -clampInDegrees.y * 0.5f,
clampInDegrees.y * 0.5f);

                transform.localRotation *= targetOrientation;

                // If there's a character body that acts as a parent to the camera
                if (characterBody)
                {
                        var yRotation = Quaternion.AngleAxis(_mouseAbsolute.x, characterBody.transform.up);
                        characterBody.transform.localRotation = yRotation;
                        characterBody.transform.localRotation *= targetCharacterOrientation;
                }
                else
                {
                        var yRotation = Quaternion.AngleAxis(_mouseAbsolute.x,
transform.InverseTransformDirection(Vector3.up));
                        transform.localRotation *= yRotation;
                }
        }
}
```

```
using UnityEngine;
using System.Collections;

public class roomMan : Photon.MonoBehaviour {

	public string verNum = "0.1";
	public string roomName = "room01";
	public string playerName = "player 420";
	public Transform spawnPoint;
	public GameObject playerPref;
	public GameObject playerPref2;
	public bool isConnected = false;
	public bool isInRoom = false;
	public GameObject[] Ais;
	public GameObject[] curAis;
	public int kD;



	public InRoomChat chat;

	public Transform[] spawnPoints;



	void Update(){
		if (isInRoom) {
			chat.enabled = true;
		} else {
			//chat.enabled = false;
		}

		curAis = GameObject.FindGameObjectsWithTag ("Ai");

		if (Input.GetKeyDown (KeyCode.E) && isInRoom == false) {
```

```
                spawnAi ();
        }
        if (PlayerPrefs.GetInt ("kills") >= 1) {
                kD = PlayerPrefs.GetInt ("kills") / PlayerPrefs.GetInt ("deaths");
        } else {
                kD = 0;
        }

        PhotonNetwork.player.SetScore (PlayerPrefs.GetInt ("kills"));
}

void Start(){

        roomName = "Room " + Random.Range (0, 999);
        playerName = "Player " + Random.Range (0, 999);
        PhotonNetwork.ConnectUsingSettings (verNum);
        Debug.Log ("Starting Connection!");



        //PlayerPrefs.SetInt ("kills", 0);
        //PlayerPrefs.SetInt ("deaths", 0);

}

public void OnJoinedLobby(){
        //PhotonNetwork.JoinOrCreateRoom (roomName, null, null);
        isConnected = true;
        Debug.Log ("Starting Server!");
}

public void OnJoinedRoom(){
        PhotonNetwork.playerName = playerName;

        GameObject[] players = GameObject.FindGameObjectsWithTag ("Player");
```

```
            foreach(GameObject pl in players){
                if(pl.name == PhotonNetwork.playerName){
                    PhotonNetwork.playerName = playerName + " " + Random.Range (0,999);
                }
            }


            isConnected = false;
            isInRoom = true;
            //spawnPlayer ();
        }


        public void spawnPlayer(string prefName){
            isInRoom = false;

            GameObject[] players = GameObject.FindGameObjectsWithTag ("Player");

            foreach(PhotonPlayer pl2 in PhotonNetwork.playerList){
                if(pl2.name == PhotonNetwork.playerName){
                    PhotonNetwork.playerName = playerName + " " + Random.Range (0,999);
                }
            }


            GameObject pl = PhotonNetwork.Instantiate (prefName, spawnPoints[Random.Range (0,
spawnPoints.Length)].position, spawnPoint.rotation, 0) as GameObject;
            pl.GetComponent<RigidbodyFPSWalker> ().enabled = true;
            pl.GetComponent<RigidbodyFPSWalker> ().fpsCam.SetActive (true);
            pl.GetComponent<RigidbodyFPSWalker> ().graphics.SetActive (false);

        }

        public void spawnAi(){
```

```
                GameObject ai1 = PhotonNetwork.Instantiate (Ais[Random.Range(0,Ais.Length)].name,
spawnPoints[Random.Range (0, spawnPoints.Length)].position, spawnPoint.rotation, 0) as GameObject;
                ai1.GetComponent<aiCon> ().isMine = true;


        }

        void OnGUI(){

                if (isConnected) {
                        Cursor.visible = true;
                        Cursor.lockState = CursorLockMode.None;
                        GUILayout.BeginArea (new Rect (Screen.width / 2 - 250, Screen.height / 2 - 250, 500,500));
                        playerName = GUILayout.TextField (playerName);
                        roomName = GUILayout.TextField (roomName);

                        if (GUILayout.Button ("Create")) {
                                PhotonNetwork.JoinOrCreateRoom (roomName, null, null);
                        }

                        foreach (RoomInfo game in PhotonNetwork.GetRoomList()) {
                                if (GUILayout.Button (game.name + " " + game.playerCount + "/" +
game.maxPlayers)) {
                                        PhotonNetwork.JoinOrCreateRoom (game.name, null, null);
                                }
                        }
                        GUILayout.EndArea ();
                }

                if (isInRoom) {
                        Cursor.visible = true;
                        Cursor.lockState = CursorLockMode.None;
                        GUILayout.BeginArea (new Rect (Screen.width / 2 - 250, Screen.height / 2 - 250, 500,500));
                        GUILayout.Box("Score: " + PhotonNetwork.player.GetScore () + " \n \nBots: x" +
curAis.Length + "\n");
```

```
                GUILayout.Box("Kills: " + PlayerPrefs.GetInt ("kills") + " | "  + "Deaths: " +
PlayerPrefs.GetInt ("deaths") + " | K/D: " + kD)  ;




                if (GUILayout.Button ("Assault")) {
                        spawnPlayer(playerPref2.name);
                }
                if (GUILayout.Button ("SWAT")) {
                        spawnPlayer(playerPref.name);
                }
                if (GUILayout.Button ("Disconnect")) {
                        PhotonNetwork.Disconnect ();
                        Application.LoadLevel (0);
                }
                if (GUILayout.Button ("Spawn 2 Bots")) {
                        spawnAi ();
                        spawnAi ();
                }
                if (GUILayout.Button ("Delete Bot")) {
                        GameObject.Find("_NETWORKSCRIPTS").GetComponent<PhotonView>().RPC
("deleteBot", PhotonTargets.AllBuffered, null);
                }
                GUILayout.EndArea ();
        }
    }



}
```