Week 4
Shooting Mechanism

Andre, M.Sc.
1604C29C – Workshop Game Development

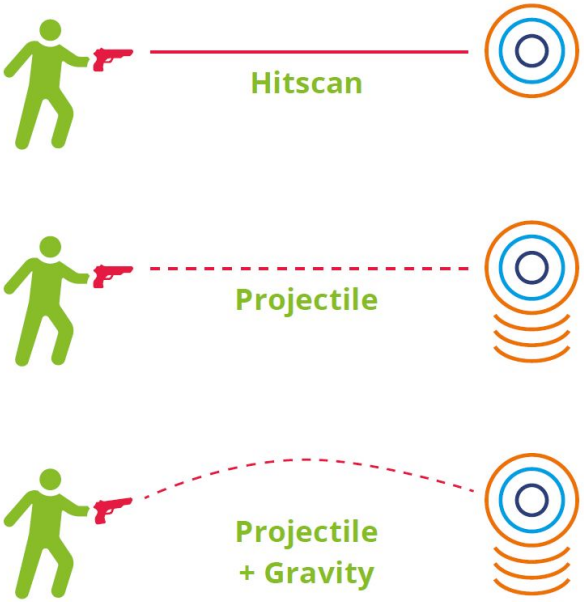
Program Game Development
Fakultas Teknik Informatika
Universitas Surabaya

Shooting Mechanism

- **Hitscan and projectile (ballistic)** methods are two ways to determine if a player's weapon hits a target in an FPS game
- Hitscan **instantly** determines whether the weapon **hits the target** by emitting a **virtual ray in the player's aim direction.** (simple and low computational)
- Projectile ballistics **simulates the physical trajectory** of a fired projectile, including **travel time and physical forces** like gravity and wind. (complex and computational expensive)

Shooting Mechanism

- Hitscan is commonly used for **modern or futuristic weapons**, while projectile ballistics is used for **realistic or historical weapons**.
- The choice between these methods depends on the game's design goals and desired gameplay experience.
- Hitscan is better suited for **fast-paced action and precision aiming**.
- Projectile ballistics is better for **realistic and strategic gameplay**



Unity Raycast

- In general, a ray is a line that extends from a point in space in a specific direction.
- In Unity, a Raycast is a function that casts a ray from a given position and direction, and returns information about any objects the ray intersects with.

```
bool Raycast(Vector3 origin, Vector3 direction, out RaycastHit hitInfo,
float maxDistance = Mathf.Infinity, int layerMask =
DefaultRaycastLayers);
```

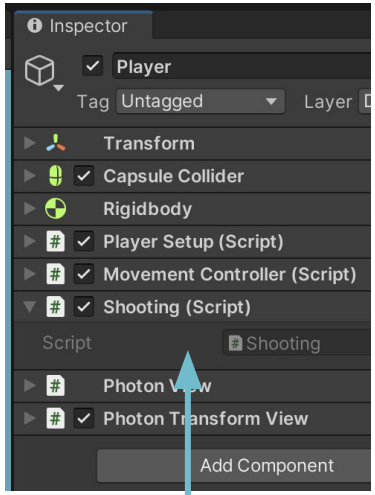
- Raycasting can be used to detect collisions, to calculate the distance between two objects, or to determine if an object is in the line of sight of another object, among other things

01 **Apply** Hitscan

Using raycast to hit enemy

Shooting Script

- create a new script called **Shooting** in the Scripts folder
- **Attach** the script into player prefab
- create a **Serializedfield Camera** reference called **fpsCamera** in the Shooting script
- the ray will be sent from the **center of the screen.**
- put an if statement inside Update method for firing when the left mouse button is clicked



Create shooting script and drag and drop into player prefab

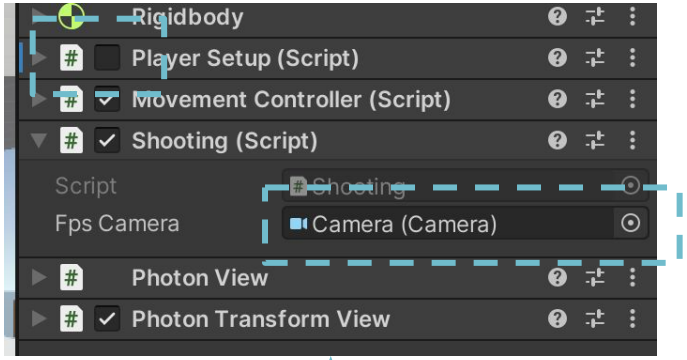
```
public class Shooting : MonoBehaviour
{
    [SerializeField] Camera fpsCamera;


    // Update is called once per frame
    void Update()
    {
        if(Input.GetButton("Fire1")) {
            RaycastHit _hit;
            Ray ray = fpsCamera.ViewportPointToRay(new Vector3(0.5f, 0.5f));

            if(Physics.Raycast(ray, out _hit,100)) {
                Debug.Log(_hit.collider.gameObject.name);
            }
        }
    }
}
```

Getting reference to camera

Create a ray in the direction at the center of screen

Detect mouse click, and call the Raycast method

The debug will show the object that obstruct the ray path



Disable the **Player Setup** script (for testing purpose) and don't forget to **attach player camera** to shooting script. Apply the prefab changes, and hit the play button

Try to shoot the building. Raycast works fine outputting on the console the object that have been hit by ray. However, it keep **hitting the same target multiple times with just single click.**

The reason of this is because we don't yet implement the fire rate to delay between shot.

Implementing Fire Rate

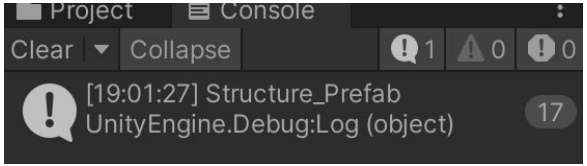
- In order to prevent continuous firing, we need to create a timer that limits how frequently the player can fire.
- To do this, we'll need to **keep track of the time since the last shot was fired.**
- If the time since the last shot is **less than a certain threshold** (the fire rate), we **won't allow the player to fire**.
- Creating a public float variable for fire rate
- Setting up a timer to delay firing based on fire rate

```
public class Shooting : MonoBehaviour
{
    [SerializeField] Camera fpsCamera;
    public float fireRate =0.3f;
    float fireTimer = 0.0f;

    void Update()
    {
        if(fireTimer < fireRate) {
            fireTimer += Time.deltaTime;
        }

        if(Input.GetButton("Fire1") && fireTimer > fireRate) {
            fireTimer = 0.0f;

            RaycastHit _hit;
            Ray ray = fpsCamera.ViewportPointT

            if(Physics.Raycast(ray, out _hit,100)) {
                Debug.Log(_hit.collider.gameObject.name);
            }
        }
    }
}
```

Create fire rate and timer variable

Increase fire timer with delta time (time difference between the current frame and the previous frame)

Prevent fire if fire timer below fire rate. This is to prevent quick succession or multiple shot at the same time



Don't forget to **enable Player Setup script**, **override the player prefab,** and **delete the player game object from game scene**

02

Taking Damage

Use Remote Procedure Call to trigger an action to other players remotely

Taking Damage Logic

- Previously, we able to hit another object with Raycast technique
- Moreover, by comparing gameobject tag to “Player” tag, we can give damage or reduce enemy health. For this purpose, we can create TakingDamage method in player script.
- The question is how can we call the TakingDamage method in enemy client?

Remote Procedure Calls

- RPCs are a way to call functions remotely, where the function is executed on a remote client or server.
- This can be useful for a variety of tasks, such as updating player positions, handling player input, or triggering game events.
- To use RPCs in Photon, first define **a method** with the **PunRPC attribute**.
- This method can then be called on any **networked object** in the game using the **PhotonView.RPC method**.
- When an RPC is called, Photon sends the **method name** and **any parameters** to the target clients or the server, where the method is executed.

Remote Procedure Calls

- To call the methods marked as PunRPC, you need a PhotonView component. Example call:

```
[PunRPC]
void ChatMessage(string a, string b)
{     Debug.Log(string.Format("ChatMessage {0} {1}", a, b));      }


PhotonView photonView = PhotonView.Get(this);
photonView.RPC("ChatMessage", RpcTarget.All, "Hello", "and greetings");
```

Targets, Buffering and Order

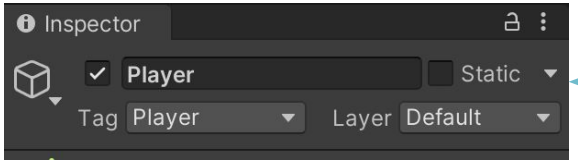
- RpcTarget can be used to define **which clients** execute an RPC
- RpcTarget values commonly used are **All** clients or **Others**. This value may followed by additional terms such as “Buffered”, “ViaServer”. I.e: “AllBuffered”, “OtherViaServer”
- “**Buffered**" –> the server will remember those RPCs and when a new player joins, it gets the RPC, even though it happened earlier.
- "**ViaServer**" –> are executed immediately by the sending client without sending the RPC through the server

Targets, Buffering and Order

- Using "ViaServer" disables the "All" shortcut, which can be useful for executing RPCs in a specific order
- An example use of "**AllViaServer**" is in a **racing game** where the first "finished" RPC call will determine the winner and subsequent calls will determine the rankings
- A specific player in the room can be targeted with an RPC using the overload with the target Player as the second parameter
- Targeting the local player will execute the RPC locally without going through the server

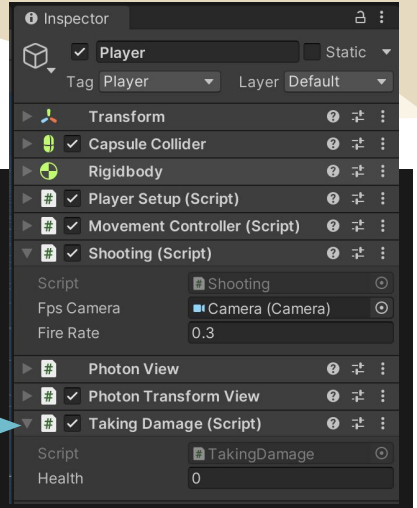
Shooting Players in Multiplayer Game

- Open player prefab
- Tag **Player** prefab with **Player tag** for identification
- Create **TakingDamage script** to hold player's **health** and **take damage**
- Don't forget to **attach** the TakingDamage script into player prefab
- Open **Shooting** script and add **if statement to check if the object being shot is a Player**
- Use **Remote Procedure Calls (RPCs)** to **apply damage** to **remote player's health**
- **PunRPC attribute** targets method to remote clients in the same room



Tag Player prefab with Player tag for identification

Lastly, don't forget to attach the Taking Damage script into player prefab




```
...
using Photon.Pun;

public class TakingDamage : MonoBehaviour
{
    public float health;

    // Start is called before the first frame update
    void Start()
    {
        health = 100f;
    }

    [PunRPC]
    public void TakeDamage(float _damage) {
        health -= _damage;
        Debug.Log(health);
    }
}
```

Set the default player health to 100

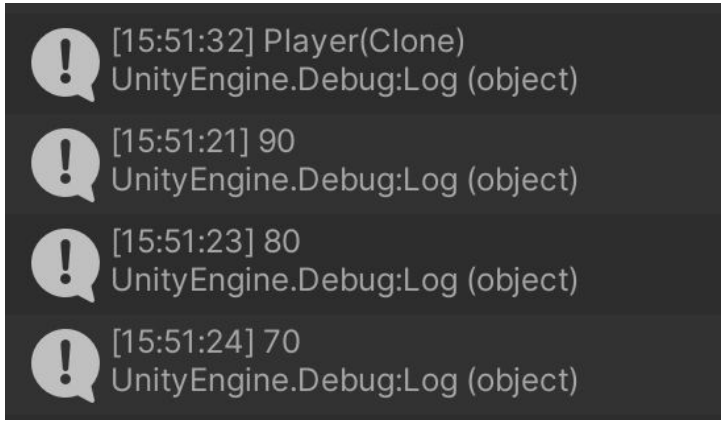
Create TakeDamage method to reduce player health. This method is identified as “PunRPC” and allowed to be called from other remote client

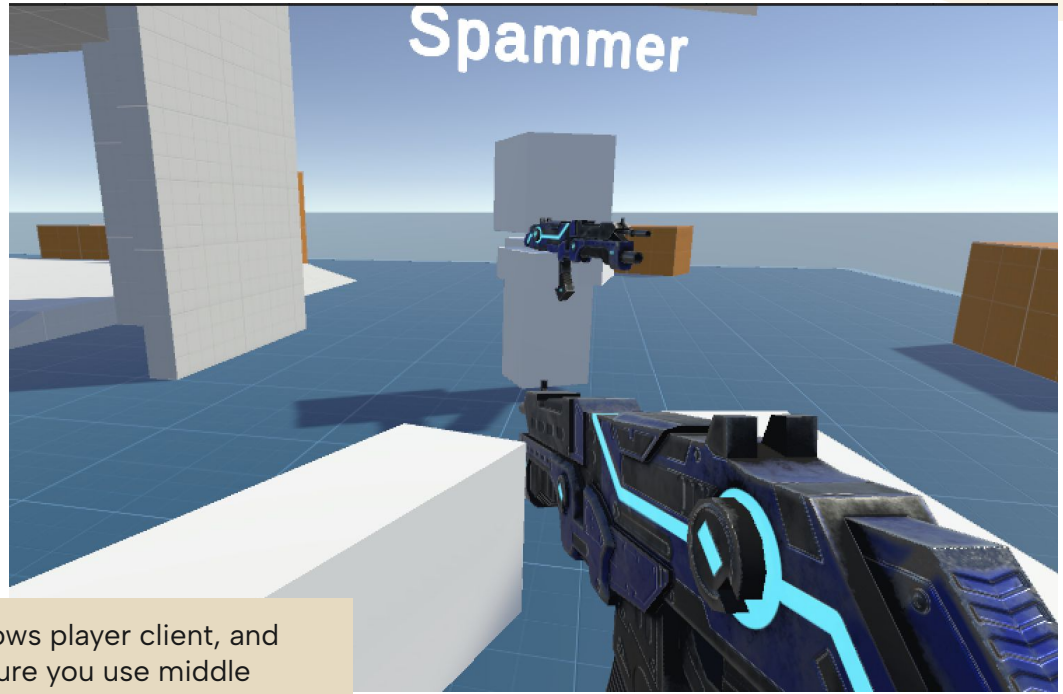
Shooting Script

```
void Update()
{
        . . .

        if(Input.GetButton("Fire1") && fireTimer > fireRate) {
            fireTimer = 0.0f;

            RaycastHit _hit;
            Ray ray = fpsCamera.ViewportPointToRay(new Vector3(

            if(Physics.Raycast(ray, out _hit,100)) {
                Debug.Log(_hit.collider.gameObject.name);
                if(_hit.collider.gameObject.CompareTag("Player") &&
                      !_hit.collider.gameObject.GetComponent<PhotonView>().IsMine) {

                    _hit.collider.gameObject.GetComponent<PhotonView>()
                          .RPC("TakeDamage",RpcTarget.AllBuffered, 10f);

                }
            }
        }
    }
```

This is to prevent damaging itself

Use Remote Procedure Calls (RPCs) to apply damage to remote player's health. The TakeDamage method is called, and amount of 10 is used as method parameter. It means the enemy health will be reduced by 10. This method uses AllBuffered.





Try it with windows player client, and shoot it. Make sure you use middle screen as crosshair. Inspect the console to read player health decreasing.

03

Updating Health

visualize the change in player health using a health bar and show how to add a crosshair to the screen

Update Player Health

- Open the **Player** prefab and open the **TakingDamage** script.
- Import **UnityEngine.UI** and create a **SerializedField Image** reference called **healthBar**.
- Create a public float reference called **startHealth** and make the health variable private.
- Set the health to **startHealth** in the Start method.
- Set the fill amount of the health bar to **health divided by startHealth** to display the health as a **percentage**.

Update Player Health

- Inside the TakeDamage method, **decrease the health** by the amount of damage.
- Set the fill amount of the health bar to **health divided by startHealth.**
- If the **health is less** than or **equal to zero**, create a **Die method**.
- Save the script and **drag and drop healthBar** to the corresponding field in **TakingDamage** script.
- Click on **Apply All** to save the changes.

Adding Crosshair

- Create **a canvas** and set its **Scale Mode as Scale With Screensize.**
- Right-click on Canvas and **create an Image called crosshair.**
- Change the sprite to **Knob** and make it **smaller**.
- Change the color of the crosshair to **red**

Taking Damage Script

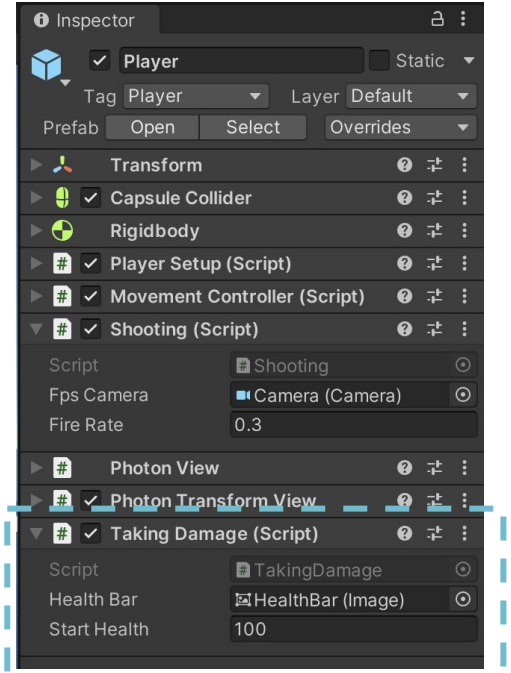
```
. . .
using UnityEngine.UI;

public class TakingDamage : MonoBehaviour
{
    [SerializeField] Image healthBar;
    private float health;
    public float startHealth =100f;

    void Start()
    {
        health = startHealth;
        healthBar.fillAmount = health/startHealth;
    }

    [PunRPC]
    public void TakeDamage(float _damage) {
        health -= _damage;
        Debug.Log(health);

        healthBar.fillAmount = health/startHealth;
        if(health < 0f) {
            Die();
        }
    }

    void Die() {

    }
}
```

Import UI library

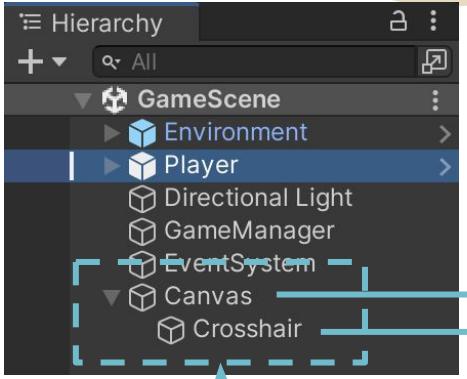
Health refer to current player health.
startHealth refer to player starting health.

This equation is to convert health into percentages. I.e: 70/100 = 0.7. Because fillAmount attribute only accept range between 0 to 1.

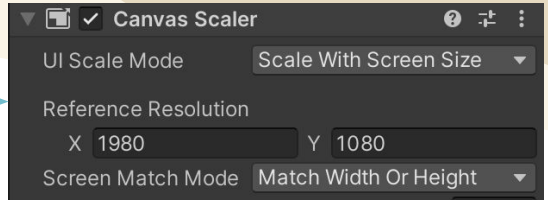
If health reach zero, then call die method



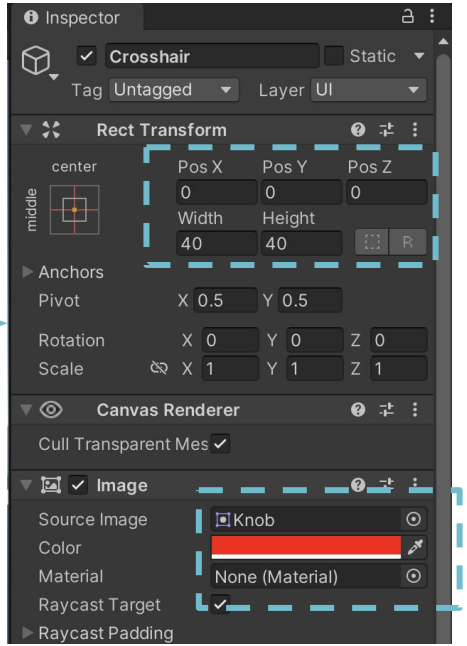
Drag and Drop the HealthBar into TakingDamage script



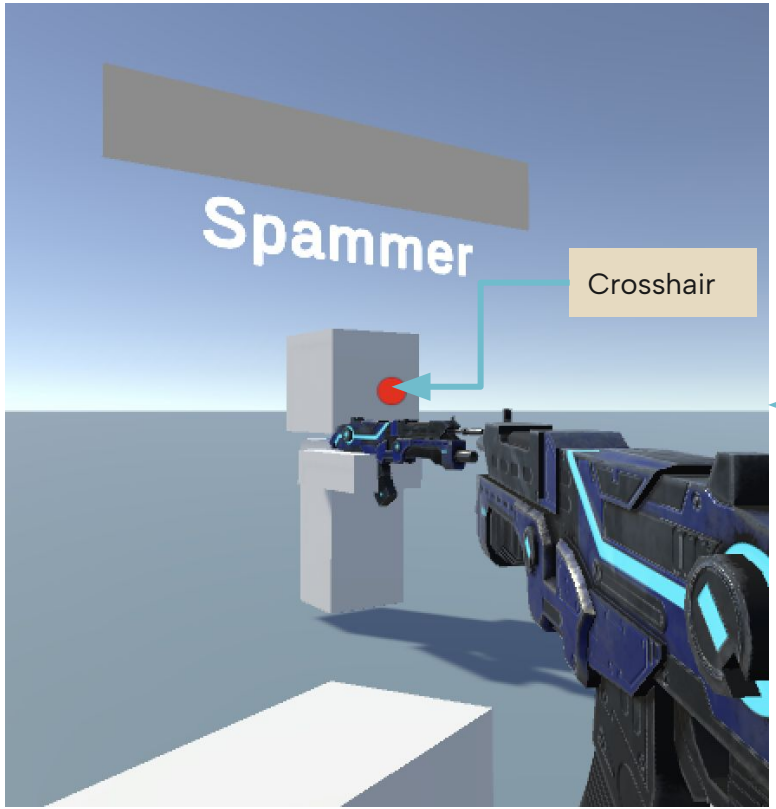
Create **canvas**, create image name it "**crosshair**"



Switch UI Scale Mode to **scale with screen size**, adjust resolution to **HD**



Make the crosshair **smaller**, use source image as "**Knob**", change color to **red**



Test with two client, and try to shoot other player. Notice the health bar is reduced.

04

Die Method

Implement singleton class and leave the room

Leaving Room

- Open the **TakingDamage** script.
- Delete Monobehaviour and type **MonobehaviourPunCallbacks**.
- Inside the Die method, **check if photonView is mine** to ensures only the player that got shot will die.
- The player will simply **leave the room** upon dying.
- This action should be managed by the **BlockyFPSGameManager**.
- Create a new method in **BlockyFPSGameManager** called "**LeaveRoom**".
- Inside that, type **PhotonNetwork.LeaveRoom().**
- This is how easy it is to leave the room in Photon.

Taking Damage Script

```
. . .

public class TakingDamage : MonoBehaviourPunCallbacks
{
      . . .
      void Die() {
            if(photonView.IsMine) {

            }
      }
}
```

Method die can be accessed by all players. Check if the character isMine to make sure the targeted player is taking damage and die.

Game Manager Script

```
public class BlockyFPSGameManager : MonoBehaviourPunCallbacks
{
      . . .
    public void LeaveRoom() {
        PhotonNetwork.LeaveRoom();
    }

    public override void OnLeftRoom() {
        SceneManager.LoadScene("LauncherScene");
    }

}
```

Player dies = leave room. Redirect the player to Launcher Scene

Singleton Class

- To call the **LeaveRoom** method from the **TakingDamage** script, use **singleton implementation**.
- A singleton class is a design pattern that **restricts the instantiation** of a class to a **single object.**
- In other words, a singleton class **allows only one instance of the class to be created** and ensures that this instance is **globally accessible**.
- Singleton is great to use for **controller classes** like **GameManager** or **AudioController**.

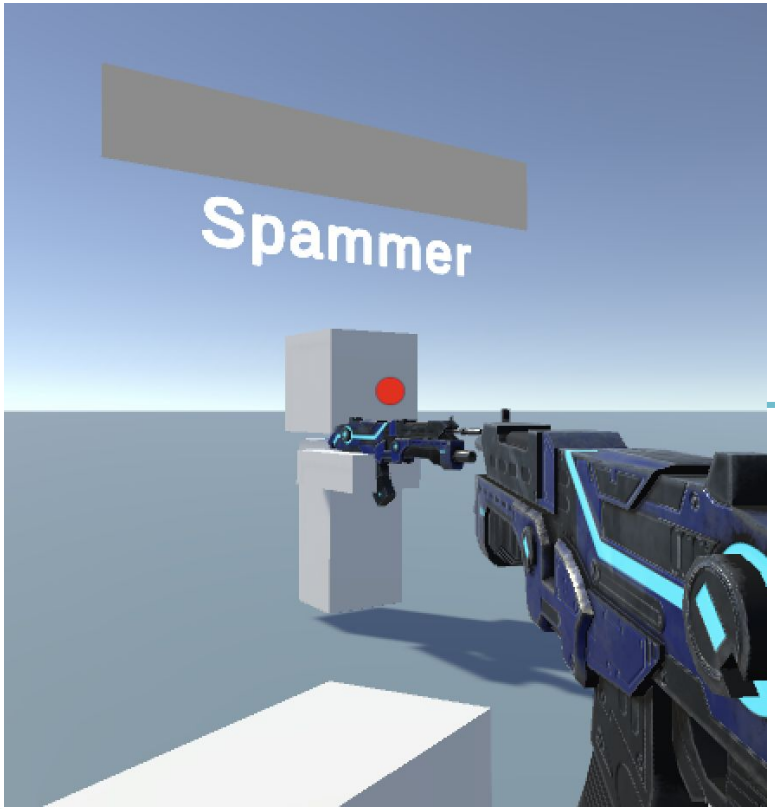
```
. . .
public class BlockyFPSGameManager : MonoBehaviourPunCallbacks
{
    [SerializeField] GameObject playerPrefab;
    public static BlockyFPSGameManager instance;

    private void Awake() {
        if(instance!=null) {
            Destroy(this.gameObject);
        } else {
            instance = this;
        }
    }
}
```

Implement singleton class in BlockyFPSManager. If a new game object (game manager) created, destroy it immediately

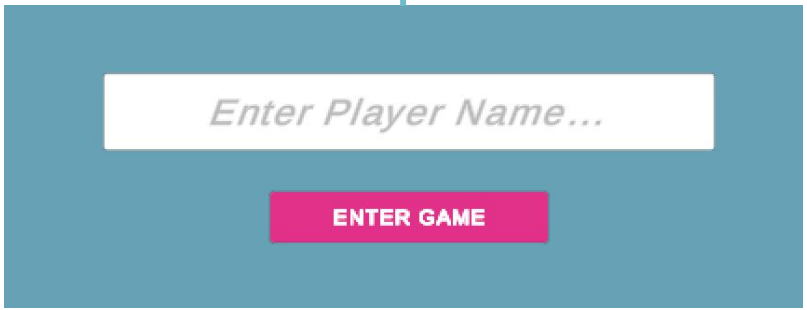
Taking Damage Script

```
. . .

void Die() {
      if(photonView.IsMine) {
            BlockyFPSGameManager.instance.LeaveRoom();
       }
}
```

Call leave room if player dies



If player dies, it goes to launcher scene



Thanks

Do you have any questions?

youremail@freepik.com / +34 654 321 432 / yourwebsite.com

slidesgo