AIController

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;
using UnityEngine.Events;

public class AIController : MonoBehaviour
{
    NavMeshAgent agent;
    PlayerCharacter character;
    Transform targetTrans;

    public UnityEvent onTankDeath;

    void Start()
    {
        // Get references to components
        character = GetComponent<PlayerCharacter>();
        agent = GetComponent<NavMeshAgent>();

        // Find the target (player) transform
        targetTrans = GameObject.FindGameObjectWithTag("Player").transform;

        // Invoke the FireControl method repeatedly with a delay of 1 second, starting after 1 second
        InvokeRepeating("FireControl", 1, 3);
    }

    void FireControl()
    {
        // Call the Fire method of the character (PlayerCharacter)
        character.Fire();
    }

    void Update()
    {
        // Set the agent's destination to the target's position
        agent.destination = targetTrans.position;

        // Rotate the AI towards the target
        transform.LookAt(targetTrans);
    }

    void Die()
```

```
    {
        // Invoke the onTankDeath event, notifying any subscribed listeners
        if (onTankDeath != null)
        {
            onTankDeath.Invoke();
        }
    }
}
```

ButtonAction

```
using UnityEngine;
using UnityEngine.UI;

public class ButtonAction : MonoBehaviour
{
    public Button quitButton;

    private void Start()
    {
        // Add a listener to the quitButton's onClick event, which will call the QuitGame method when clicked
        quitButton.onClick.AddListener(QuitGame);
    }

    private void QuitGame()
    {
        // Check if the game is running in the Unity Editor
        #if UNITY_EDITOR
            // Set the Unity Editor's "isPlaying" property to false, effectively stopping the game
            UnityEditor.EditorApplication.isPlaying = false;
        #else
            // Quit the application (only works in standalone builds)
            Application.Quit();
        #endif
    }
}
```

ButtonStart

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
```

```
using UnityEngine.UI;

public class ButtonStart : MonoBehaviour
{
    public Button startButton;
    public string nextSceneName;

    private void Start()
    {
        // Add a listener to the startButton's onClick event, which will call the StartGame method when clicked
        startButton.onClick.AddListener(StartGame);
    }

    private void StartGame()
    {
        // Load the scene specified by the nextSceneName variable using SceneManager.LoadScene
        SceneManager.LoadScene(nextSceneName);
    }
}
```

GameManager

```
using UnityEngine;
using UnityEngine.Events;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    public GameObject player;
    public string gameEndSceneName;
    public string gameWinSceneName;

    private PlayerCharacter playerCharacter;
    private AIController[] aiControllers;

    private void Start()
    {
        // Get the PlayerCharacter component from the player GameObject
        playerCharacter = player.GetComponent<PlayerCharacter>();
        // Add a listener to the onPlayerDeath event of the playerCharacter
        playerCharacter.onPlayerDeath.AddListener(OnPlayerDeath);

        // Find all AIControllers in the scene
```

```
        aiControllers = FindObjectsOfType<AIController>();

        // Add a listener to the onTankDeath event of each AIController
        foreach (AIController aiController in aiControllers)
        {
            aiController.onTankDeath.AddListener(OnTankDeath);
        }
    }

    private void OnPlayerDeath()
    {
        // Load the gameEndScene when the player dies
        SceneManager.LoadScene(gameEndSceneName);
    }

    private void OnTankDeath()
    {
        bool allTanksDead = true;

        // Check if any AIController is still alive
        foreach (AIController aiController in aiControllers)
        {
            if (aiController != null)
            {
                // If an AIController is found, set allTanksDead to false and break out of the loop
                allTanksDead = false;
                break;
            }
        }

        // If all tanks are dead, load the gameWinScene
        if (allTanksDead)
        {
            SceneManager.LoadScene(gameWinSceneName);
        }
    }
}
```

PlayerCharacter

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

```
public class PlayerCharacter : MonoBehaviour
{
    public float speed;
    public float turnSpeed;
    public ParticleSystem explosionParticles;

    public Rigidbody shell;
    public Transform muzzle;

    public float launchForce = 10;
    public AudioSource shootAudioSource;

    public float health;
    float healthMax;
    bool isAlive;

    public Slider healthSlider;
    public Image healthFillImage;
    public Color healthColorFull = Color.green;
    public Color HealthColorNull = Color.red;

    CharacterController cc;

    bool attacking = false;
    public float attackTime;

    Animator animator;

    public UnityEvent onPlayerDeath;

    void Start()
    {
        animator = GetComponentInChildren<Animator>();
        cc = GetComponent<CharacterController>();
        healthMax = health;
        isAlive = true;
        RefreshHealthHUD();
        explosionParticles.gameObject.SetActive(false);
    }

    // Take damage and check if the player dies
    public void TakeDamage(float amount)
    {
        health -= amount;
```

```
    RefreshHealthHUD();
    if (health <= 0f && isAlive)
    {
        Death();
    }
}

// Update the health HUD slider and fill image
public void RefreshHealthHUD()
{
    healthSlider.value = health;
    healthFillImage.color = Color.Lerp(HealthColorNull, healthColorFull, health / healthMax);
}

// Handle the player's death
public void Death()
{
    isAlive = false;
    explosionParticles.transform.parent = null;
    explosionParticles.gameObject.SetActive(true);
    ParticleSystem.MainModule mainModule = explosionParticles.main;
    Destroy(explosionParticles.gameObject, mainModule.duration);
    gameObject.SetActive(false);

    // Invoke the onPlayerDeath event
    if (onPlayerDeath != null)
    {
        onPlayerDeath.Invoke();
    }
}

// Move the player character
public void Move(Vector3 v)
{
    if (!isAlive) return;
    if (attacking) return;
    Vector3 movement = v * speed;
    cc.SimpleMove(movement);
    if (animator)
    {
        animator.SetFloat("Speed", cc.velocity.magnitude);
    }
}

// Rotate the player character towards the given direction
```

```
    public void Rotate(Vector3 lookDir)
    {
        var targetPos = transform.position + lookDir;
        var characterPos = transform.position;

        characterPos.y = 0;
        targetPos.y = 0;

        Vector3 faceToDir = targetPos - characterPos;
        Quaternion faceToQuat = Quaternion.LookRotation(faceToDir);
        Quaternion slerp = Quaternion.Slerp(transform.rotation, faceToQuat, turnSpeed *
Time.deltaTime);

        transform.rotation = slerp;
    }

    // Fire a shell from the muzzle position
    public void Fire()
    {
        if (!isAlive) return;
        if (attacking) return;

        Rigidbody shellInstance = Instantiate(shell, muzzle.position, muzzle.rotation) as
Rigidbody;
        shellInstance.velocity = launchForce * muzzle.forward;
        shootAudioSource.Play();

        if (animator)
        {
            animator.SetTrigger("Attack");
        }

        attacking = true;
        Invoke("RefreshAttack", attackTime);
    }

    // Refresh the attack state
    void RefreshAttack()
    {
        attacking = false;
    }
}
```

PlayerController

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO.Ports;

public class PlayerController : MonoBehaviour
{
    PlayerCharacter character;
    SerialPort serialPort;

    public string portName = "/dev/cu.usbmodem1101";
    public int baudRate = 9600;
    public int distanceThreshold = 10;

    private bool canFire = false;

    void Start()
    {
        character = GetComponent<PlayerCharacter>();

        // Create a new SerialPort instance with the specified port name and baud rate
        serialPort = new SerialPort(portName, baudRate);
        // Open the serial port
        serialPort.Open();
    }

    void Update()
    {
        if (serialPort.IsOpen && serialPort.BytesToRead > 0)
        {
            // Read the data from the serial port
            string data = serialPort.ReadLine();
            data = data.Trim(); // Remove any leading or trailing whitespaces

            // Log the received Arduino data
            Debug.Log("Arduino Data: " + data);

            // Parse the received data as an integer
            if (int.TryParse(data, out int arduinoValue))
            {
                // Set the canFire variable based on the received Arduino value
                if (arduinoValue == 1)
                {
                    canFire = true;
                }
```

```
            else
            {
                canFire = false;
            }
        }
    }

    // Fire if the canFire variable is true
    if (canFire)
    {
        character.Fire();
    }

    // Read input from the keyboard or joystick for character movement and rotation
    var h = Input.GetAxis("Horizontal");
    var v = Input.GetAxis("Vertical");
    character.Move(new Vector3(h, 0, v));

    var lookDir = Vector3.forward * v + Vector3.right * h;
    if (lookDir.magnitude != 0)
    {
        character.Rotate(lookDir);
    }
}

void OnDestroy()
{
    // Close the serial port when the script or game object is destroyed
    if (serialPort != null && serialPort.IsOpen)
    {
        serialPort.Close();
    }
}
}
```

Shell

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Shell : MonoBehaviour
{
    public float lifeTimeMax = 2f;
    public AudioSource explosionAudioSource;
```

```
    public ParticleSystem explosionParticles;
    public float explosionRadius;
    public float explosionForce = 1000f;
    public float damageMax = 100f;
    public LayerMask damageMask;
    public bool isRotate = false;

    void Start()
    {
        // Destroy the shell object after a certain lifetime
        Destroy(gameObject, lifeTimeMax);

        // If rotation is enabled, add torque to the shell's rigidbody
        if (isRotate)
        {
            GetComponent<Rigidbody>().AddTorque(transform.right * 1000);
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        // Find all colliders within the explosion radius and matching the damage mask
        Collider[] colliders = Physics.OverlapSphere(transform.position, explosionRadius,
damageMask);

        // Loop through each collider
        foreach (var collider in colliders)
        {
            // Check if the collider's GameObject has a PlayerCharacter component
            var targetCharacter = collider.GetComponent<PlayerCharacter>();
            if (targetCharacter)
            {
                // Calculate and apply damage to the player character based on the distance
from the explosion
                targetCharacter.TakeDamage(CalculateDamage(collider.transform.position));
            }
        }

        // Detach the explosion particles from the shell object, play the explosion audio, and
start the explosion particle effect
        explosionParticles.transform.parent = null;
        explosionAudioSource.Play();
        explosionParticles.Play();
```

```
        // Get the main module of the explosion particles and destroy the particle system after its duration
        ParticleSystem.MainModule mainModule = explosionParticles.main;
        Destroy(explosionParticles.gameObject, mainModule.duration);

        // Destroy the shell object
        Destroy(gameObject);
    }

    float CalculateDamage(Vector3 targetPosition)
    {
        // Calculate the distance between the target position and the shell's position
        var distance = (targetPosition - transform.position).magnitude;

        // Calculate the damage modifier based on the distance from the explosion
        var damageModify = (explosionRadius - distance) / explosionRadius;

        // Calculate the actual damage based on the modifier and the maximum damage value
        var damage = damageModify * damageMax;

        // Ensure the minimum damage is 2 to prevent very low damage values
        return Mathf.Max(2f, damage);
    }
}
```