implementation-in-action

T# Code Examples

The code examples illustrate how to write scripts in T#. You will notice that each of these scripts uses only available wrappers and avoids any Unity C# functionality that is not supported in T# as is stated in the section on Unsupported Functionalities in T#.

Example 1: Skybox Rotation

```
 // The code below rotates the SkyBox per frame
using System;
using System.Collections;
using Terra.Studio;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;
using UnityEngine;

public class SkyboxRotator : StudioBehaviour
{
    private float m_fRotationSpeed = 1.5f;
    private float m_fRotationValue;

    // Gets called every frame
    private void Update()
    {
        m_fRotationValue += m_fRotationSpeed * Time.deltaTime;
        RenderSettings.skybox.SetFloat("_Rotation", m_fRotationValue);
    }
}
```

Example 2: Emission Color Change

```
 // Resets the emission Color to Black in Start
//Then changes the emission color of a tile back when the player touches it
using System;
using System.Collections;
using Terra.Studio;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;
using UnityEngine;

public class TileColorChanger : StudioBehaviour
{
    private Color m_refEmissionColor;
    private bool m_bColorHasChanged;
    // Gets called at the start of the lifecycle of the GameObject
    private void Start()
    {
        m_refEmissionColor = (this.gameObject.GetComponent(typeof(Renderer)) as Renderer).material.GetCol
or("_EmissionColor");
```

```
        m_bColorHasChanged = false;
        (this.gameObject.GetComponent(typeof(Renderer)) as Renderer).material.SetColor("_EmissionColor",
Color.black);
    }

    void OnTriggerEnter(Collider col)
    {
        if (!m_bColorHasChanged && StudioController.CheckIfController(col.gameObject) != null && m_refEmi
ssionColor != null)
        {
            (this.gameObject.GetComponent(typeof(Renderer)) as Renderer).material.SetColor("_EmissionColo
r", m_refEmissionColor);
            m_bColorHasChanged = true;
        }
    }
}
```

Example 3: Playing a Random VFX

```
// Plays a random VFX when the player reaches a CheckPoint or when Dying
using System;
using System.Collections;
using Terra.Studio;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;
using TerraStudio.TerraSharp.Collections;
using Unity.Mathematics;
using UnityEngine;

public class MiscStuff : StudioBehaviour
{
    private TerraList m_refDeathParticles;
    private TerraList m_refCheckpointCrossedParticles;

    // Gets called at the start of the lifecycle of the GameObject
    private void Start()
    {
        m_refCheckpointCrossedParticles = new TerraList();
        m_refDeathParticles = new TerraList();
        for (int i = 0; i < 10; i++)
        {
            GameObject toAdd = GetGameObjectVariable("GoodParticle" + i);
            if (toAdd != null)
            {
                m_refCheckpointCrossedParticles.Add(toAdd);
            }
            else
            {
                break;
            }
        }
        for (int i = 0; i < 10; i++)
        {
```

```
            GameObject toAdd = GetGameObjectVariable("KillParticle" + i);
            if (toAdd != null)
            {
                m_refDeathParticles.Add(toAdd);
            }
            else
            {
                break;
            }
        }
    }

    // Gets called whenever a broadcast is triggered by Behaviours or Other T# scripts
    public override void OnBroadcasted(string x)
    {
        Debug.Log("Broadcast received : " + x);
        if (string.Equals(x, "PlayerKilled"))
        {
            PlayRandomSadParticle();
        }
        else if (x.StartsWith("CP"))
        {
            PlayRandomHappyParticle();
        }
    }

    private void PlayRandomSadParticle()
    {
        GameObject kill = Instantiate(GetRandomLoseParticle(), StudioController.GetMyController().GetLoca
tor("headwear_l1_loc"));
        kill.transform.localPosition = new Vector3(0f, 0.01f, 0f);
        kill.transform.localScale *= 0.01f;
        StartCoroutine(DestroyAfter3Seconds(kill));
    }

    private void PlayRandomHappyParticle()
    {
        GameObject kill = Instantiate(GetRandomWinParticle(), StudioController.GetMyController().GetLocat
or("headwear_l1_loc"));
        kill.transform.localPosition = new Vector3(0f, 0.01f, 0f);
        kill.transform.localScale *= 0.01f;
        StartCoroutine(DestroyAfter3Seconds(kill));
    }

    private GameObject GetRandomWinParticle()
    {
        return m_refCheckpointCrossedParticles[UnityEngine.Random.Range(0, m_refCheckpointCrossedParticle
s.Count)] as GameObject;
    }

    private GameObject GetRandomLoseParticle()
    {
        return m_refDeathParticles[UnityEngine.Random.Range(0, m_refDeathParticles.Count)] as GameObject;
    }
```

```
    IEnumerator DestroyAfter3Seconds(GameObject a_refObj)
    {
        yield return new WaitForSeconds(3.0f);
        if (a_refObj != null)
        {
            Destroy(a_refObj);
        }
    }
}
```

Example 4: Camera & Light Change

```
 // On reaching the last checkpoint, Moves the Camera to victory pose
// It then changes the lights so the character looks good
using System;
using System.Collections;
using Terra.Studio;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;
using UnityEngine;

public class EndCinematics : StudioBehaviour
{
    private GameObject m_refLight1;
    private GameObject m_refLight2;
    // Gets called at the start of the lifecycle of the GameObject
    private void Start()
    {
       m_refLight1 = GetGameObjectVariable("Light1");
       m_refLight2 = GetGameObjectVariable("Light2");
    }

    // Gets called whenever a broadcast is triggered by Behaviours or Other T# scripts
    public override void OnBroadcasted(string x)
    {
        if (string.Equals(x, "CP20"))
        {
            StartCoroutine(MoveCameraForWin());
        }
    }

    private IEnumerator MoveCameraForWin()
    {
        Transform cameraTarget = StudioController.GetMyController().GetControllerCamera().transform.paren
t;
        Vector3 currentPos = cameraTarget.localPosition;
        Vector3 posToGoTo = new Vector3(0f, -3.7f, -1.1f);

        Vector3 currentRot = cameraTarget.localEulerAngles;
        Vector3 rotToGoTo = new Vector3(74.709f, 180f, 0);

        m_refLight1.transform.localEulerAngles = new Vector3(m_refLight1.transform.localEulerAngles.x, m_
```

```
refLight1.transform.localEulerAngles.y-180f, m_refLight1.transform.localEulerAngles.z);
        m_refLight2.transform.localEulerAngles = new Vector3(m_refLight2.transform.localEulerAngles.x, m_
refLight2.transform.localEulerAngles.y-180f, m_refLight2.transform.localEulerAngles.z);

        float time = 0f;

        do
        {
            cameraTarget.localPosition = Vector3.Lerp(currentPos, posToGoTo, time);
            cameraTarget.localEulerAngles = Vector3.Lerp(currentRot, rotToGoTo, time);
            time += Time.deltaTime;
            yield return new WaitForEndOfFrame();
        } while (time <= 1f);
        cameraTarget.localPosition = posToGoTo;
        cameraTarget.localEulerAngles = rotToGoTo;
        Broadcast("DoTheDance");
    }
}
```

Example 5: The Top Down Action Controller

This is one of the controllers available as a readymade package in Terra Studio and has been entirely written in T#. To access this Select `File` in the Main Toolbar on the top and import the Top Down Action package. The controller ties 7 scripts together together, creating a system where:

- Enemies are detected and tracked automatically
- The player's weapon aims and shoots at tracked enemies
- Bullets are managed and reloaded as needed
- Enemies take damage and are destroyed when their health depletes

This creates an automated combat system where the player's character can engage enemies within range without direct input, handling detection, aiming, shooting, and reloading automatically.

The controller is represented by the `StudioController` class, uses these scripts to create a cohesive weapon system. Here's an overview of how the controller utilizes these scripts to work overall:

1. **Initialization**: The `TerraBootstrap.cs` script acts as the entry point. It initializes the main components: the `TerraWeaponController.cs`, `TerraAutoAim.cs` system and the `TerraEnemy.cs` with references to the controller and auto-aim system
2. **Weapon Setup**: The `TerraWeaponController.cs` handles the initial weapon setup:, instantiates the weapon object, sets up the `TerraProjectileWeapon.cs` component and configures the player's animator for weapon poses
3. **Enemy Detection and Tracking**: The `TerraEnemy.cs` script checks if the player is within its detection radius. If so, it requests the `TerraAutoAim.cs` to track the enemy. `TerraAutoAim.cs` manages the tracking, including player rotation towards the enemy
4. Shooting Mechanism: `TerraWeaponController.cs` continuously checks for tracked enemies in its Update method. If an enemy is tracked and in line of sight, it triggers the shooting process. `TerraProjectileWeapon.cs` handles the actual shooting logic. It checks if shooting is allowed (based on fire rate and magazine status) and it uses `TerraMagazine.cs` to spawn bullets and manage ammo count
5. Bullet Behavior: `TerraBullet.cs` scripts control individual bullet behavior. They handle movement and collision detection. On collision with an enemy, they trigger damage calculation
6. Enemy Interaction: When a bullet hits an enemy, the `TerraEnemy.cs` script reduces the enemy's health and destroys the enemy if health reaches zero
7. Weapon Management: `TerraProjectileWeapon.cs` manages overall weapon state. It tracks magazine count and initiates reloads when necessary. `TerraMagazine.cs` handles individual magazine logic: manages bullet counts and reloading for each magazine

8. Player Feedback: `TerraAutoAim.cs` provides visual feedback by drawing the detection radius. `TerraWeaponController.cs` manages weapon animations through the player's animator

Here is a detailed description of each.

`TerraAutoAim.cs`

Receives in a request to track an enemy and once tracked forces the player's model to look only towards that enemy

```
 // TerraAutoAim.cs
// Handles auto-aiming functionality for the player

using UnityEngine;
using System.Collections;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;

public class TerraAutoAim : StudioBehaviour
{
    private float targetSwitchDelay = 1f;
    private float radius;
    private float turnRate = 5f;
    private float lineOfSightAngle = 30f;
    private Material meshMaterial;
    private Transform modelTr;
    private bool wasTrackingInLastFrame;
    private Transform currentLockedEnemy;
    private float cachedRadius;
    private Mesh cachedMesh;
    private Coroutine coroutine;
    private StudioController controller;

    public Transform CurrentLockedEnemy => currentLockedEnemy;

    // Initializes variables from Studio variables
    private void Start()
    {
        targetSwitchDelay = GetFloatVariable("TargetSwitchDelay");
        radius = GetFloatVariable("Radius");
        turnRate = GetFloatVariable("TurnRate");
        lineOfSightAngle = GetFloatVariable("LineOfSightAngle");
        meshMaterial = (GetGameObjectVariable("MeshReference").GetComponent(typeof(MeshRenderer)) as Mesh
Renderer).material;
        modelTr = transform;
    }

    // Initializes the controller reference
    public void Init(StudioController controller)
    {
        this.controller = controller;
    }

    // Attempts to track a new enemy
    // Returns true if tracking is successful, false otherwise
    public bool TryTrackEnemy(Transform newEnemy)
```

```
    {
        if (!newEnemy || !modelTr)
        {
            return false;
        }
        var distance = Vector3.Distance(newEnemy.position, modelTr.position);
        if (coroutine != null || distance > radius)
        {
            return false;
        }
        if (currentLockedEnemy && distance > GetTrackedEnemyDistanceDelta())
        {
            return false;
        }
        Untrack(currentLockedEnemy);
        currentLockedEnemy = newEnemy;
        TogglePlayerMovementRotation(false);
        coroutine = StartCoroutine(TrackCoroutine());
        return true;
    }

    // Checks if the currently tracked enemy is within line of sight
    // Returns true if enemy is in sight, false otherwise
    public bool IsEnemyInLineOfSight()
    {
        if (!currentLockedEnemy)
        {
            return false;
        }
        var targetDir = currentLockedEnemy.transform.position - modelTr.position;
        targetDir.y = 0f;
        var angle = Vector3.Angle(modelTr.forward, targetDir);
        return angle < lineOfSightAngle;
    }

    // Stops tracking the specified enemy
    public void Untrack(Transform enemy)
    {
        if (!currentLockedEnemy)
        {
            return;
        }
        if (currentLockedEnemy == enemy)
        {
            ResetStoredValues();
        }
    }

    // Coroutine to handle the delay between target switches
    private IEnumerator TrackCoroutine()
    {
        yield return new WaitForSeconds(targetSwitchDelay);
        coroutine = null;
    }
```

```
// Stops the tracking coroutine if it's running
private void HaltTrackCoroutine()
{
    if (coroutine == null)
    {
        return;
    }
    StopCoroutine(coroutine);
    coroutine = null;
}

// Toggles the player's ability to rotate
private void TogglePlayerMovementRotation(bool status)
{
    controller.TogglePlayerRotation(status);
}

// Resets all tracking-related values
private void ResetStoredValues()
{
    currentLockedEnemy = null;
    HaltTrackCoroutine();
    TogglePlayerMovementRotation(true);
}

// Updates tracking and mesh drawing every frame
private void Update()
{
    DrawMesh();
    if (!currentLockedEnemy)
    {
        if (wasTrackingInLastFrame)
        {
            ResetStoredValues();
        }
        wasTrackingInLastFrame = false;
        return;
    }
    LockOnToEnemy();
    CheckCurrentDelta();
    wasTrackingInLastFrame = true;
}

// Checks if the tracked enemy is still within range
private void CheckCurrentDelta()
{
    var distance = GetTrackedEnemyDistanceDelta();
    if (distance > radius || distance < 0f)
    {
        ResetStoredValues();
    }
}
```

```
    // Calculates the distance to the currently tracked enemy
    private float GetTrackedEnemyDistanceDelta()
    {
        if (!currentLockedEnemy)
        {
            return float.PositiveInfinity;
        }
        return Vector3.Distance(currentLockedEnemy.position, modelTr.position);
    }

    // Rotates the player model to face the tracked enemy
    private void LockOnToEnemy()
    {
        var targetDir = currentLockedEnemy.transform.position - modelTr.position;
        targetDir.y = 0f;
        var step = Time.deltaTime * turnRate;
        var newDir = Vector3.RotateTowards(modelTr.forward, targetDir, step, 0.0f);
        modelTr.rotation = Quaternion.LookRotation(newDir);
    }

    // Draws the detection radius mesh
    private void DrawMesh()
    {
        if (cachedRadius != radius) GenerateMesh();
        var position = transform.position;
        position.y += 0.1f;
        Graphics.DrawMesh(cachedMesh, position, Quaternion.Euler(new Vector3(-180f, 0f, 0f)), meshMateria
l, 0);
    }

    // Generates a new mesh for the detection radius
    private void GenerateMesh()
    {
        if (!cachedMesh) Destroy(cachedMesh);
        var mesh = CreateDisc(radius, 32);
        cachedMesh = mesh;
        cachedRadius = radius;
    }

    // Creates a disc-shaped mesh for the detection radius
    private Mesh CreateDisc(float radius, int segments)
    {
        var mesh = new Mesh();
        var vertices = new Vector3[segments + 1];
        var triangles = new int[segments * 3];
        vertices[0] = Vector3.zero;
        var angleStep = 360f / segments;
        for (int i = 1; i <= segments; i++)
        {
            var angle = angleStep * i * Mathf.Deg2Rad;
            vertices[i] = new Vector3(Mathf.Cos(angle) * radius, 0, Mathf.Sin(angle) * radius);
            if (i < segments)
            {
                triangles[(i - 1) * 3] = 0;
```

```
                triangles[(i - 1) * 3 + 1] = i;
                triangles[(i - 1) * 3 + 2] = i + 1;
            }
        }
        triangles[(segments - 1) * 3] = 0;
        triangles[(segments - 1) * 3 + 1] = segments;
        triangles[(segments - 1) * 3 + 2] = 1;
        mesh.vertices = vertices;
        mesh.triangles = triangles;
        mesh.RecalculateNormals();
        mesh.name = $"Disc {radius} {segments}";
        return mesh;
    }
}
```

`TerraBootstrap.cs`

This is a global script which just lets other scripts to have a common reference. Similar to any other bootstrap functionality

```
 using System.Collections;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;

public class TerraBootstrap : StudioBehaviour
{
    private TerraWeaponController weaponController;
    private TerraAutoAim autoAim;

    // Starts the setup process
    private void Start()
    {
        StartCoroutine(WaitAndSetup());
    }

    // Waits for a frame and then sets up component references
    private IEnumerator WaitAndSetup()
    {
        yield return null;
        var controller = GetGameObjectVariable("Controller");
        weaponController = controller.GetComponent(typeof(TerraWeaponController)) as TerraWeaponControlle
r;
        autoAim = controller.GetComponent(typeof(TerraAutoAim)) as TerraAutoAim;
        StartCoroutine(WaitForController());
    }

    // Waits for the controller to be available and initializes components
    private IEnumerator WaitForController()
    {
        StudioController controller = null;
        while (controller == null)
        {
            yield return null;
            controller = StudioController.GetMyController();
        }
```

```
        weaponController.Init(controller);
        autoAim.Init(controller);
        var enemyGO = GetGameObjectVariable("Enemy");
        var enemy = enemyGO.GetComponent(typeof(TerraEnemy)) as TerraEnemy;
        enemy.Init(controller, autoAim);
    }
}
```

`TerraBullet.cs`

This script is Attached on a bullet travels in forward direction until it collides with any collider

```
 // Attached on a bullet that travels in forward direction
using UnityEngine;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;

public class TerraBullet : StudioBehaviour
{
    public float speed = 15f;
    public float destroyAfter = 10f;
    public float damageToDeal = 1f;
    private Rigidbody rb;

    private Vector3 cachedForward;

    // Initializes bullet properties and components
    private void Start()
    {
        speed = GetFloatVariable("Speed");
        destroyAfter = GetFloatVariable("DestroyAfter");
        damageToDeal = GetFloatVariable("DamageToDeal");
        rb = GetComponent(typeof(Rigidbody)) as Rigidbody;
        cachedForward = transform.forward;
    }

    // Updates the bullet's lifetime and destroys it if necessary
    private void Update()
    {
        if (destroyAfter <= 0f)
        {
            Destroy(gameObject);
        }
        else
        {
            destroyAfter -= Time.deltaTime;
        }
    }

    // Moves the bullet forward at a constant speed
    private void FixedUpdate()
    {
        if (speed != 0)
        {
```

```
            rb.velocity = cachedForward * speed;
        }
    }

    // Handles collision with objects and destroys the bullet
    private void OnCollisionEnter(Collision other)
    {
        var controller = StudioController.CheckIfController(other.gameObject);
        if (controller != null) return;
        Destroy(gameObject);
    }
}
```

`TerraEnemy.cs`

When player enters certain radius, this script requests auto aim to track the current enemy and on hit by bullet deduces it's own health. If the health is less than or equal to , then destroys the player

```
 using UnityEngine;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;

public class TerraEnemy : StudioBehaviour
{
    private float radius;
    private float totalHealth;
    private TerraAutoAim autoAim;
    private StudioController controller;

    // Initializes enemy properties
    private void Start()
    {
        radius = GetFloatVariable("Radius");
        totalHealth = GetFloatVariable("TotalHealth");
    }

    // Initializes controller and auto-aim references
    public void Init(StudioController controller, TerraAutoAim autoAim)
    {
        this.controller = controller;
        this.autoAim = autoAim;
    }

    // Checks if the player is within range and requests tracking
    private void Update()
    {
        if (controller == null)
        {
            return;
        }
        var delta = Vector3.Distance(controller.GetPlayerPosition(), transform.position);
        if (delta <= radius)
        {
            autoAim.TryTrackEnemy(transform);
```

```
        }
    }

    // Handles collision with bullets, updates health, and destroys enemy if necessary
    private void OnCollisionEnter(Collision other)
    {
        if (other == null || other.gameObject == null)
        {
            return;
        }
        var script = other.gameObject.GetComponent(typeof(TerraBullet));
        if (script == null)
        {
            return;
        }
        var bullet = script as TerraBullet;
        if (bullet == null)
        {
            return;
        }
        totalHealth -= bullet.damageToDeal;
        Debug.Log("Health left: " + totalHealth);
        if (totalHealth <= 0)
        {
            Destroy(gameObject);
        }
    }
}
```

`TerraMagazine.cs`

This holds a record of how many can exist per magazine in a weapon, and when shoot is invoked it spawns a bullet

```
 using UnityEngine;
using Terra.Studio.Exposed;

public class TerraMagazine : StudioBehaviour
{
    private GameObject bulletPrefab;
    private Transform bulletSpawnLoc;
    private int totalBulletCount;
    private int currentBulletCount;

    // Initializes magazine properties and reloads
    private void Start()
    {
        bulletPrefab = GetGameObjectVariable("BulletPrefab");
        bulletSpawnLoc = GetGameObjectVariable("BulletSpawnLoc").transform;
        totalBulletCount = GetIntVariable("TotalBulletCount");
        ReloadMag();
    }

    // Checks if there are bullets available to shoot
    public bool CanShoot()
```

```
    {
        return currentBulletCount > 0;
    }

    // Spawns a bullet, sets its direction, and decreases the bullet count
    public void Shoot(Vector3 dir)
    {
        var bullet = Instantiate(bulletPrefab);
        bullet.transform.position = bulletSpawnLoc.position;
        bullet.gameObject.layer = LayerMask.NameToLayer("Bullet");
        bullet.transform.forward = dir;
        bullet.SetActive(true);
        currentBulletCount--;
    }

    // Resets the bullet count to full
    public void ReloadMag()
    {
        currentBulletCount = totalBulletCount;
    }
}
```

`TerraProjectileWeapon.cs`

This Holds a record of how many magazines can exist for this projectile weapon and attempts to reload when the magazine has run out of bullets

```
 using UnityEngine;
using System.Collections;
using Terra.Studio.Exposed;

public class TerraProjectileWeapon : StudioBehaviour
{
    public int totalMagazines;
    public int firingRate;
    public float reloadTime;
    private Coroutine reloadCoroutine;
    private int currentActiveMagazines;
    private TerraMagazine currentMagazine;
    private float lastBulletShotTime;

    // Initializes weapon properties and gets the magazine component
    private void Start()
    {
        totalMagazines = GetIntVariable("TotalMagazines");
        firingRate = GetIntVariable("FiringRate");
        reloadTime = GetFloatVariable("ReloadTime");
        StartCoroutine(WaitAndGetMagazine());
    }

    // Coroutine to wait for magazine component to be available
    private IEnumerator WaitAndGetMagazine()
    {
        yield return null;
```

```
        currentMagazine = GetComponent(typeof(TerraMagazine)) as TerraMagazine;
    }

    // Sets up the weapon with initial values
    public void Equip()
    {
        currentActiveMagazines = totalMagazines;
        lastBulletShotTime = float.NegativeInfinity;
    }

    // Attempts to shoot in the specified direction
    public void Shoot(Vector3 dir)
    {
        if (currentMagazine == null) return;
        if (currentActiveMagazines <= 0) return;
        OnAttemptedToShoot(dir);
    }

    // Handles the actual shooting process, including rate of fire and reloading
    private void OnAttemptedToShoot(Vector3 dir)
    {
        if (!CanShoot()) return;
        if (reloadCoroutine != null) return;
        if (currentMagazine.CanShoot())
        {
            currentMagazine.Shoot(dir);
            lastBulletShotTime = Time.time;
        }
        //Check => so that we can do an early reload
        if (!currentMagazine.CanShoot())
        {
            reloadCoroutine ??= StartCoroutine(DoReload());
        }
    }

    // Checks if enough time has passed to allow another shot
    private bool CanShoot()
    {
        var delta = Time.time - lastBulletShotTime;
        return delta >= 1f / firingRate;
    }

    // Coroutine to handle the reload delay
    private IEnumerator DoReload()
    {
        yield return new WaitForSeconds(reloadTime);
        reloadCoroutine = null;
        ReloadWeapon();
    }

    // Reloads the weapon by resetting the magazine
    public void ReloadWeapon()
    {
        currentActiveMagazines--;
```

```
        currentMagazine.ReloadMag();
    }
}
```

`TerraWeaponController.cs`

This is an Initiator script which sets up the projectile and player's animation

```
 using UnityEngine;
using System.Collections;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;

public class TerraWeaponController : StudioBehaviour
{
    //==Assign from Inspector==
    private float delayBetweenChecks = 1f;
    private Transform dir;

    //==Animation==
    private const string POSE_LAYER_NAME = "PoseLayer";
    private const string POSE_NAME = "one_handgun_one_hand_idle";

    private TerraAutoAim autoAim;
    private TerraProjectileWeapon Weapon;
    private GameObject weaponObj;
    private StudioController controller;

    // Initializes controller properties
    private void Start()
    {
        delayBetweenChecks = GetFloatVariable("DelayBetweenChecks");
        weaponObj = GetGameObjectVariable("Weapon");
        dir = GetGameObjectVariable("Dir").transform;
    }

    // Initializes the controller reference and starts setup
    public void Init(StudioController controller)
    {
        this.controller = controller;
        StartCoroutine(Setup());
    }

    // Coroutine to set up weapon and animator
    private IEnumerator Setup()
    {
        Transform weaponLoc = null;
        Animator animator = null;
        while (weaponLoc == null)
        {
            weaponLoc = controller.GetLocatorEnumBased(StudioController.PlayerLocEnum.ShootIKLoc);
            yield return new WaitForSeconds(delayBetweenChecks);
        }
        yield return StartCoroutine(SetupWeapon(weaponLoc));
```

```
        yield return StartCoroutine(SetupAnimator(animator));
        autoAim = gameObject.GetComponent(typeof(TerraAutoAim)) as TerraAutoAim;
    }

    // Sets up the weapon object and its components
    private IEnumerator SetupWeapon(Transform weaponLoc)
    {
        var weaponObj = Instantiate(this.weaponObj, weaponLoc);
        weaponObj.transform.localPosition = new Vector3(-0.00044f, 0.00031f, 0.00124f);
        weaponObj.transform.localRotation = Quaternion.Euler(new Vector3(0.025f, -20.058f, -0.005f));
        weaponObj.transform.localScale = Vector3.one * 0.0095f;
        yield return null; //Mandatory wait to fetch the component, only available in the next frame
        Weapon = weaponObj.GetComponent(typeof(TerraProjectileWeapon)) as TerraProjectileWeapon;
        if (Weapon != null)
        {
            Weapon.Equip();
        }
    }

    // Sets up the animator and plays the initial pose
    private IEnumerator SetupAnimator(Animator animator)
    {
        while (animator == null)
        {
            animator = gameObject.GetComponentInChildren(typeof(Animator)) as Animator;
            yield return new WaitForSeconds(delayBetweenChecks);
        }
        if (animator != null)
        {
            var layerIndex = animator.GetLayerIndex(POSE_LAYER_NAME);
            animator.Play(POSE_NAME, layerIndex);
            animator.SetLayerWeight(layerIndex, 1f);
        }
    }

    // Attempts to automatically shoot at tracked enemies each frame
    private void Update()
    {
        AttemptAutoShoot();
    }

    // Checks if an enemy is tracked and in sight, then attempts to shoot
    private void AttemptAutoShoot()
    {
        if (!autoAim || !autoAim.CurrentLockedEnemy) return;
        if (!autoAim.IsEnemyInLineOfSight()) return;
        Shoot();
    }

    // Triggers the weapon to shoot
    public void Shoot()
    {
        Weapon.Shoot(dir.forward);
```

```
    }
}
```