# <u>MLH Local Hack Day December 2017 - Unity Workshop Notes</u>

What will be covered:

- Editor
- C# Scripting
- Animation (Mecanim)
- Physics
- Audio
- Basic AI
- UI

Game Features:

- Top-down view
- Plot: Cowboy abducted by evil machines must escape
- Being discovered means certain death
- Basic Inventory System
- Interacting with an environment (e.g. pressing buttons, opening doors)

## Contents

# What will be covered

## Introduction to the Editor

- Editor Basics
- Typical C# Script Structure

## Creating a character controller

- Setting up an existing character for animation system (Mecanim).
    - Creating Animator Controller asset
    - Adding parameters
    - Adding animations to Mecanim
    - Using blend shapes and sub-machine states
- Writing scripts
    - LocomotionController
    - JumpController
    - PlayerController
    - HealthController

## Creating Collectables

- Creating an Apple Collectable
    - AppleItem
- Creating a Key Collectable and a Door
    - KeyCollector
    - KeyItem
    - DoorController

## Creating UI

- Canvas
- Positioning elements
- HealthUIRender

## Creating Robot AI

- Baking a NavMesh
- Setting up a NavMeshAgent
- RobotController

## Adding Audio

- Setting up AudioMixer
- Adding Audio Sources
- AudioManager

# Scripts

## Player Camera

```
using UnityEngine;

// This attribute tells Unity that this component requires a Camera component to be present.
[RequireComponent(typeof(Camera))]
public class PlayerCamera : MonoBehaviour
{
    // Uncheck this if you don't want Camera to track a Transform target.
    public bool IsTracking = true;

    // Transform of a GameObject which a Camera will track.
    public Transform Target;

    // Determines how smooth the camera will transition to a new position
    public float SmoothValue = 2.5f;

    // Affects the final position of a camera
    public Vector3 CameraOffset = new Vector3(0, 10, -10);

    // Final position which is calculated at the end of this frame, this value is what camera is going to lerp to
    private Vector3 _targetPos;

    /*
     * LateUpdate() is called right after the Update().
     * Cases where code affects the position/rotation of a Camera should always be implemented in
     * LateUpdate() because it tracks objects that might have moved inside Update().
    */
    private void LateUpdate()
    {
        // An elegant way to say that Camera should not be affected if IsTracking is set to false.
        if (!IsTracking) return;

        // We determine a new Camera position by taking target's position and adding offset on top of it.
        _targetPos = new Vector3(
            Target.position.x + CameraOffset.x,
            CameraOffset.y,
            Target.position.z + CameraOffset.z
        );

        // We smoothly change Camera's position to a position we calculated earlier.
        transform.position = Vector3.Lerp(
            transform.position,
            _targetPos,
            Time.deltaTime * SmoothValue
        );
    }
}
```

## Animator Controller

```
using UnityEngine;

[RequireComponent(typeof(Animator))]
// Acts as an abstraction between controllers and animations
public class AnimatorController : MonoBehaviour
{
    // An animator component
    private Animator animator;

    private void Start()
    {
        animator = GetComponent<Animator>();
    }

    // Sets the movement value which will be smoothyl interpolated over time
    public void SetMovement(float val)
    {
        float newVal = Mathf.Lerp(animator.GetFloat("speed_mov"), val, Time.deltaTime * 5);
        animator.SetFloat("speed_mov", newVal);
    }

    // Sets the rotation value which will be smoothyl interpolated over time
    public void SetRotation(float val)
    {
        animator.SetFloat("speed_rot", val);
    }

    // Triggers a jump animation
    public void SetJump()
    {
        animator.SetTrigger("jump");
    }

    // Updates the grounded animation based on a given value
    public void SetGrounded(bool val)
    {
        animator.SetBool("grounded", val);
    }
}
```

## Apple Item

```
using UnityEngine;

// Handles picking up apples
public class AppleItem : MonoBehaviour
{
    // Amount of health it restores
    public float HealthValue = 10;

    // Call this event when it enters other collider
    private void OnTriggerEnter(Collider other)
    {
        // Find collider's health controller
        HealthController health = other.gameObject.GetComponent<HealthController>();

        // If it's not null, this means that the entity collided with an apple is Player
        if (health)
        {
            // Give it health if possible
            if (health.GiveHealth(10))
            {
                // Destroy itself if giving health was successful
                Destroy(this.gameObject);
            }
        }
    }
}
```

## Audio Manager

```
using UnityEngine;

// Manages all audio in the game. The way it works is that it subscribes to objects and plays soun
ds when necessary.
public class AudioManager : MonoBehaviour
{
    private AudioSource audio;                    // Audio Source component

    public AudioClip keyPickup;                   // Audio clip which will be played when key is picked up

    private KeyItem[] keys;                       // Used to store all keys that exist within scene

    void Awake()
    {
        audio = GetComponent<AudioSource>();     // Finds an Audio Source
        keys = FindObjectsOfType<KeyItem>();     // Finds all keys in the scene. Avoid using this m
ethod often as large scene will make such operations slower

        for (int i = 0; i < keys.Length; i++)
        {
            keys[i].KeyPickup += OnKeyPickup;    // Subscribe to the pickup event of all keys
        }
    }

    private void OnKeyPickup(KeyItem item)
    {
        item.KeyPickup -
= OnKeyPickup;          // You must always unsubscribe from methods whose objects are about to be
destroyed
        audio.PlayOneShot(keyPickup);            // Play an audio clip of pickup sound
    }
}
```

## Door Controller

```csharp
using UnityEngine;

// Used to control doors
public class DoorController : MonoBehaviour
{
    public bool IsLocked = true;        // returns true if this door is locked

    private Rigidbody _body;            // rigidbody of this door

    private void Start()
    {
        _body = GetComponent<Rigidbody>();
    }

    // Gets executed when the door opens
    private void OnTriggerEnter(Collider other)
    {
        if (IsLocked)
        {
            // if the door is locked, check if collided entity has a KeyCollector
            KeyCollector _collector = other.gameObject.GetComponent<KeyCollector>();

            if (_collector)
            {
                // try unlocking the door
                if (_collector.UnlockDoor())
                {
                    UnlockDoor();       // if that entity has a key then open the door
                }
            }
        }
    }

    // Unlocks the door
    private void UnlockDoor()
    {
        IsLocked = false;       // indicate that the door is now unlocked
        // Make the door non-kinematic, allowing entities to pass through
        _body.isKinematic = false;
    }
}
```

## Game Manager

```
using UnityEngine;
using UnityEngine.SceneManagement;

// Manages the whole game. In this case, it simply restarts the scene when player dies.
public class GameManager : MonoBehaviour
{
    public HealthController PlayerHealth;           // Player's health

    private void Awake()
    {
        PlayerHealth.EntityDead += OnPlayerDead;    // Subscribe to player's OnDeadEvent
    }

    // This code will be called once player dies
    private void OnPlayerDead(HealthController player)
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex); // Reload a current scene
    }
}
```

## Health Controller

```
using UnityEngine;

public class HealthController : MonoBehaviour
{
    // Returns true if this entity is alive
    public bool IsAlive
    {
        get { return Health > 0; }
    }

    [Header("Health")]
    public float Health = 100;          // Current health
    public float MaxHealth = 100;       // Max Health

    [Header("Armor")] public float Armor = 0;       // Current armor
    public float MaxArmor = 100;                    // Max Armor

    // Specifies how much damage does it block. E.g, if real damage is 10 and ArmorBlockFactor is 
0.5, then resultant damage will be 5.
    public float ArmorBlockFactor = 0.85f;

    // Make this entity take damage. This is where it decides whether or not reduce armor or Healt
h
    public void TakeDamage(float amount)
    {
        float newDamage = DamageArmor(amount);          // calculate the leftover damage
        DamageHealth(newDamage);                        // reduce players' health with leftover da
mage
    }

    private void Start()
    {
        // Call these events in order to update the UI right when the game starts.
        OnHealthChanged(Health);
        OnArmorChanged(Armor);
    }

    // Grants health to an entity. Returns false if its health is full
    public bool GiveHealth(float amount)
    {
        if (Health < MaxHealth)                     // if the health is less than its max health
        {
            float nHealth = Health + amount;

            if (nHealth > MaxHealth)                // if the new health is higher than max health
            {
                Health = MaxHealth;                 // set health to max health to avoid overheal
            }
            else
            {
                Health = nHealth;                   // update entity health
            }

            OnHealthChanged(Health);                // tell subscribers that health has changed

            return true;
        }
        else
        {
            // return false to say that health was not needed.
            // this is useful in cases where healthpacks will not be destroyed if entity has full 
health
            return false;
        }
    }

    // Damages armor. Returns the damage which is left after armor has been withstood damage
    private float DamageArmor(float amount)
    {
        // Damage which wil be dealt to the player's health after block factor is done
        float leftOverDamage = amount;

        if (Armor > 0)          // if armor is not broken
        {
            float nArmor = Armor;       // place current armor amount to temp variable
            nArmor -= amount;           // deduct amount from that armor

            if (nArmor >= 0)                    // if recent damages did not break the armor
            {
                Armor = nArmor;                 // update the actual Armor amount
                leftOverDamage = amount - (amount * ArmorBlockFactor);          // calculate the l
eftover damage
            }
            else if (nArmor < 0)                // if the new amount is less than 0
            {
                Armor = 0;                      // set armor to 0 to avoid negative numbers
            }

            OnArmorChanged(Armor);          // call the event to make subscribers aware of changes
        }

        // If armor was broken then full amount of damage is give to player
        return leftOverDamage;
    }

    // Damages the health of the entity by a given amount
    private void DamageHealth(float amount)
    {
        float nHealth = Health - amount;        // stores the new health amount in a temp variable


        if (nHealth > 0)                        // if entity is still alive after the damage given
        {
            Health = nHealth;                   // update the health
            OnHealthChanged(Health);            // call the event to make subscribers aware of cha
nges
        }
        else                                    // if entity doesn't survive the damage
        {
            Health = 0;                         // set health to 0 in to avoid negative amount

            OnHealthChanged(Health);
            OnEntityDead(this);                 // tell subscribers that this entity just died
        }
    }

    // These are used to handle various events like health changes
    public delegate void EntityDeadEvent(HealthController controller);
    public delegate void HealthChangedEvent(float newHealth);
    public delegate void ArmorChangedEvent(float newArmor);
    public delegate void DamageTakenEvent(float amount);

    public event EntityDeadEvent EntityDead;
    public event HealthChangedEvent HealthChanged;
    public event ArmorChangedEvent ArmorChanged;

    private void OnEntityDead(HealthController controller)
    {
        if (EntityDead != null)
        {
            EntityDead.Invoke(controller);
        }
    }

    private void OnHealthChanged(float newHealth)
    {
        if (HealthChanged != null)
        {
            HealthChanged.Invoke(newHealth);
        }
    }

    private void OnArmorChanged(float newArmor)
    {
        if (ArmorChanged != null)
        {
            ArmorChanged.Invoke(newArmor);
        }
    }
}
```

## Health UI Renderer

```csharp
using UnityEngine;
using UnityEngine.UI;

// Used to render the Health UI of the player.
public class HealthUIRender : MonoBehaviour
{
    public Text HealthLabel;            // Label text of Health
    public Text ArmorLabel;             // Label text of Armor

    public HealthController _health;    // HealthController of the player

    // Called when the game starts
    private void Awake()
    {
        // Subscribe to player's health and armor changes events
        _health.HealthChanged += SetHealthText;
        _health.ArmorChanged += SetArmorText;
    }

    // Updates player's Armor UI
    private void SetHealthText(float amount)
    {
        HealthLabel.text = string.Format("Health: {0}", (int)amount);
    }

    // Updates player's Armor UI
    private void SetArmorText(float amount)
    {
        ArmorLabel.text = string.Format("Armor: {0}", (int)amount);
    }
}
```

## Jump Controller

```
using UnityEngine;

public class JumpController : MonoBehaviour
{
    public float jumpForce = 2.5f;          // specifies the jump strength

    // specifies how much will this speed be affected while airborne
    public float AirSpeedFactor = 0.2f;

    // used by other controllers to have its speed affected by jumping
    public float JumpSpeedFactor
    {
        get { return IsGrounded() ? 1 : AirSpeedFactor; }
    }

    private Rigidbody _body;            // rigidbody of this entity
    private Collider _collider;         // collider of this entity

    public AnimatorController _anim;    // handles animation

    public bool grounded;       // returns true if the entity is grounded


    private void Start()
    {
        _body = GetComponent<Rigidbody>();
        _collider = GetComponent<Collider>();
    }

    private void Update()
    {
        //grounded = IsGrounded();
        _anim.SetGrounded(IsGrounded());
    }

    // Makes this instance jump
    public void Jump()
    {
        if (IsGrounded())
        {
            _body.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
            _anim.SetJump();
        }
    }

    // how long is the trigger which checks if entity is grounded or not. the longer the ray the earlier this
 entity becomes "grounded"
    public float isGroundedRayLength = 0.1f;

    public bool IsGrounded()            // checks if thsi entity is grounded
    {
        Vector3 originPos = transform.position;
        originPos.y = _collider.bounds.min.y + 0.1f;         // draws a raycast from the collider of this ent
ity
        float rayLength = isGroundedRayLength + 0.1f;

        bool v = Physics.Raycast(originPos, Vector3.down, rayLength);

        return v;
    }
}
```

## Key Collector

```csharp
using UnityEngine;

// Handles the collection of keys and unlocking doors
public class KeyCollector : MonoBehaviour
{
    public bool HasKey = false;

    // Gives a key to this entity. Returns false if this entity already has a key
    public bool GiveKey()
    {
        if (!HasKey)
        {
            HasKey = true;
            return true;
        }

        return false;
    }

    // Unlocks a door. Returns false if it was unable to unlock a door
    public bool UnlockDoor()
    {
        if (HasKey)
        {
            HasKey = false;
            return true;
        }

        return false;
    }
}
```

## Key Item

```
using UnityEngine;

// Handles the pickup of a Key
public class KeyItem : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        KeyCollector key = other.gameObject.GetComponent<KeyCollector>();

        if (key)
        {
            if (key.GiveKey())
            {
                OnKeyPickup(this);
                Destroy(this.gameObject);
            }
        }
    }

    public delegate void OnKeyPickupEvent(KeyItem key);

    public event OnKeyPickupEvent KeyPickup;

    protected void OnKeyPickup(KeyItem key)
    {
        if (KeyPickup != null)
        {
            KeyPickup.Invoke(key);
        }
    }
}
```

## Locomotion Controller

```
using UnityEngine;

// Handles the movement and rotation of an entity
internal class LocomotionController : MonoBehaviour
{
    public float walkSpeed = 5;         // speed when an entity walks
    public float sprintSpeed = 5;       // speed when an entity runs

    public float lookSpeed = 5;         // speed at which entity rotates to a given direction

    private Vector3 moveDirection;      // current move direction
    private Vector3 finalVelocity;      // calculated velocity of this entity
    private Quaternion _targetRot = new Quaternion();           // calculated final rotation of this entity

    // Assumed that the child object is an animated mesh
    public Transform childObj;          // child object of this entity

    private Rigidbody _body;            // rigidbody component of this entity

    public bool IsRunning;              // returns true if this entity is currently running

    public AnimatorController _anim;        // AnimatorController
    private JumpController _jumpControl;    // Jump component

    public bool CanMove = true;             // Returns true if this entity can move. Overradable by other com
ponents

    private float TargetSpeed               // Speed at which this entity should be moving
    {
        get
        {
            if (IsRunning)
            {
                return sprintSpeed * _jumpControl.JumpSpeedFactor;
            }

            return walkSpeed * _jumpControl.JumpSpeedFactor;
        }
    }

    private float CurrentSpeedClamped   // Returns this entity's speed which is clamped between 0 and 1
    {
        get { return Mathf.Clamp(_body.velocity.magnitude, 0, TargetSpeed); }
    }

    private void Start()
    {
        _body = GetComponent<Rigidbody>();
        _jumpControl = GetComponent<JumpController>();
    }

    private void FixedUpdate()
    {
        // This is what actually moves the object.
        _body.velocity = Vector3.Lerp(_body.velocity, finalVelocity, Time.deltaTime * 5f);
    }

    public void Rotate(Vector2 dir)
    {
        // Get the horizontal and vertical movements from input

        Vector3 movement = new Vector3(dir.x, 0, dir.y);

        // Calculate a final rotation based on inputs that player enters.
        if (movement != Vector3.zero)
        {
            _targetRot = Quaternion.LookRotation(movement);
        }

        // Smoothly change the rotation of the child object
        childObj.rotation = Quaternion.Slerp(childObj.rotation, _targetRot, Time.deltaTime * lookSpeed);
    }

    // Moves the player in a specified direction.
    public void Move(Vector2 DIR)
    {
        // Set the direction in which this entity will move
        moveDirection = new Vector3(DIR.x * TargetSpeed, 0,
            DIR.y * TargetSpeed);

        if (CanMove)        // if this entity can move...
        {
            finalVelocity = moveDirection;      // set its final velocity

            if (_anim != null)
            {
                _anim.SetMovement(CurrentSpeedClamped);         // update entity animation
            }
        }
        else
        {
            finalVelocity = Vector3.zero;           // set the final velocity to 0

            if (_anim != null)
            {
                _anim.SetMovement(0);               // update entity animation to stop moving
            }
        }
    }
}
```

## Player Camera

```csharp
using UnityEngine;

// This attribute tells Unity that this component requires a Camera component to be present.
[RequireComponent(typeof(Camera))]
public class PlayerCamera : MonoBehaviour
{
    // Uncheck this if you don't want Camera to track a Transform target.
    public bool IsTracking = true;

    // Transform of a GameObject which a Camera will track.
    public Transform Target;

    // Determines how smooth the camera will transition to a new position
    public float SmoothValue = 2.5f;

    // Affects the final position of a camera
    public Vector3 CameraOffset = new Vector3(0, 10, -10);

    // Final position which is calculated at the end of this frame, this value is what camera is g
oing to lerp to
    private Vector3 _targetPos;

    /*
     * LateUpdate() is called right after the Update().
     * Cases where code affects the position/rotation of a Camera should always be implemented in
     * LateUpdate() because it tracks objects that might have moved inside Update().
    */

    private void LateUpdate()
    {
        // An elegant way to say that Camera should not be affected if IsTracking is set to false.
        if (!IsTracking) return;

        // We determine a new Camera position by taking target's position and adding offset on top
 of it.
        _targetPos = new Vector3(
            Target.position.x + CameraOffset.x,
            CameraOffset.y,
            Target.position.z + CameraOffset.z
        );

        // We smoothly change Camera's position to a position we calculated earlier.
        transform.position = Vector3.Lerp(
            transform.position,
            _targetPos,
            Time.deltaTime * SmoothValue
        );
    }
}
```

## Player Controller

```
using UnityEngine;

// This basically tells Unity that this script requires a different component in order to operate
correctly.
// A required component is automatically added to GameObject if such script is attached to it
[RequireComponent(typeof(LocomotionController))]

// This script handles user's input and controls the player
public class PlayerController : MonoBehaviour
{
    private LocomotionController _controller;
    private JumpController _jumpController;

    private void Awake()
    {
        _controller = GetComponent<LocomotionController>();
        _jumpController = GetComponent<JumpController>();
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            _jumpController.Jump();             // Make the player jump when "Space" is pressed
        }

        _controller.IsRunning = Input.GetKey(KeyCode.LeftShift);            // Make player run whe
n "Left Shift" is being held

        // Get the movement direction from WASD and Arrow keys
        Vector2 MoveDir = new Vector2(Input.GetAxis("Horizontal"), Input.GetAxis("Vertical"));

        _controller.Move(MoveDir);              // move player in a specified direction
        _controller.Rotate(MoveDir);            // make player look in a specified direction
    }
}
```

## Robot Controller

```
using UnityEngine;
using UnityEngine.AI;

[RequireComponent(typeof(LocomotionController))]
[RequireComponent(typeof(NavMeshAgent))]

// Handles the Robot AI
public class RobotController : MonoBehaviour
{
    public Transform FollowTarget;              // target which it will follow and attack
    private NavMeshAgent _agent;                // NavMeshAgent of this component
    private LocomotionController _controller;   // Robot's controller class
    private HealthController targetHealth;      // Target's HealthController class.

    public float MaxFollowDistance = 15;        // Specifies how far away should the target be to
stop pursuing it

    public float DamageAmount = 5;              // Specifies how much damage will this robot deal
to its target

    // Returns the distance which remains between itself and target
    private float RemainingDistance
    {
        get { return (FollowTarget.position - transform.position).magnitude; }
    }

    // Finds components and attaches to them. Basically, setting up. Better instead of manually at
taching each component
    private void Awake()
    {
        _agent = GetComponent<NavMeshAgent>();
        _agent.isStopped = true;
        _controller = GetComponent<LocomotionController>();

        targetHealth = FollowTarget.gameObject.GetComponent<HealthController>();

        path = new NavMeshPath();
    }

    private NavMeshPath path;               // the path this robot will take (being recalculated e
very frame)
    private Vector3 moveDir;                // direction in which this robot currently moves

    public float AttackCooldown = 5;        // cooldown (in seconds) between attacks
    private float currentCooldown = 0;      // current cooldown (updates every frame)

    private bool IsTargeting                // Returns true if this robot is currently in pursue o
f a target
    {
        get
        {
            if (TooFar())
            {
                return false;
            }
            else if (DestinationReached())
            {
                return false;
            }
            else
            {
                return true;
            }
        }
    }

    // Returns true if this robot can attack its target
    private bool CanAttack
    {
        get { return DestinationReached() && currentCooldown >= AttackCooldown; }
    }

    // Returns true if this robot is too far away from its target
    private bool TooFar()
    {
        return RemainingDistance >= MaxFollowDistance;
    }

    // Returns true if this robot has reached its target and is within its range
    private bool DestinationReached()
    {
        return RemainingDistance < _agent.stoppingDistance;
    }

    // Executed each frame
    private void Update()
    {
        _controller.CanMove = (IsTargeting);            // robot can move unless it's not pursing
its target

        // calculate a path base don target's current position
        _agent.CalculatePath(new Vector3(FollowTarget.position.x, 0, FollowTarget.position.z), pat
h);

        if (path.corners.Length == 1)           // if robot can walk up to its target with no obst
acles to avoid
        {
            // set its move direction towards its target
            moveDir = (FollowTarget.position - transform.position).normalized;
        }
        else if (path.corners.Length > 1)       // if robot has not pass at least one abstracle
        {
            // set its move direction towards next path corner
            moveDir = (path.corners[1] - transform.position).normalized;
        }

        // move and rotate robot based on directions calculated
        _controller.Move(new Vector2(moveDir.x, moveDir.z));
        _controller.Rotate(new Vector2(moveDir.x, moveDir.z));

        AttackTarget();     // attacks target if within its reach
    }

    private void AttackTarget()
    {
        if (CanAttack)
        {
            targetHealth.TakeDamage(DamageAmount);          // deal damage to its target if within
 its reach
```

```
            currentCooldown = 0;                            // reset cooldown to 0
        }
        else
        {
            if (currentCooldown < AttackCooldown)           // if attack is on cooldown...
            {
                currentCooldown += Time.deltaTime;          // ...then increase it
            }
        }
    }

    // This is used entirely for debugging purposes only. Useful if you want to see which path wil
l this robot take.
    // Only visible if Robot is currently pursuing its target
    private void OnDrawGizmos()
    {
        if (_controller == null || path == null)
            return;

        if (!IsTargeting)
            return;

        for (int i = 0; i < (path.corners.Length - 1); i++)
        {
            if (i == 0)
            {
                Gizmos.color = Color.red;
            }
            else if (i == path.corners.Length - 1)
            {
                Gizmos.color = Color.magenta;
            }
            else
            {
                Gizmos.color = Color.yellow;
            }

            Gizmos.DrawSphere(path.corners[i], 0.05f);

            Gizmos.color = Color.white;
            Gizmos.DrawLine(path.corners[i], path.corners[i + 1]);
        }

        if (path.corners.Length > 1)
        {
            Vector3 startPos = new Vector3(transform.position.x, 1.5f, transform.position.z);
            Vector3 endPos = startPos + (FollowTarget.position - path.corners[0]);

            Gizmos.color = Color.green;
            Gizmos.DrawLine(startPos, endPos);
        }
        else
        {
            Gizmos.DrawLine(transform.position, transform.position + Vector3.up * 10);
        }
    }
}
```

## GameObject Rotator

```
using UnityEngine;

// Used to rotate a said GameObject in a given direction
public class RotatingEffect : MonoBehaviour
{
    public float Speed = 10f;       // rotation speed
    public Vector3 Directions;      // Directions in which it will rotate

    private void Update()
    {
        // Smoothyl rotate the GameObject
        transform.Rotate(Directions, Speed * Time.deltaTime);
    }
}
```