Motores de videojuegos 2023

Práctica 2 – Enunciado completo

Sesión 1

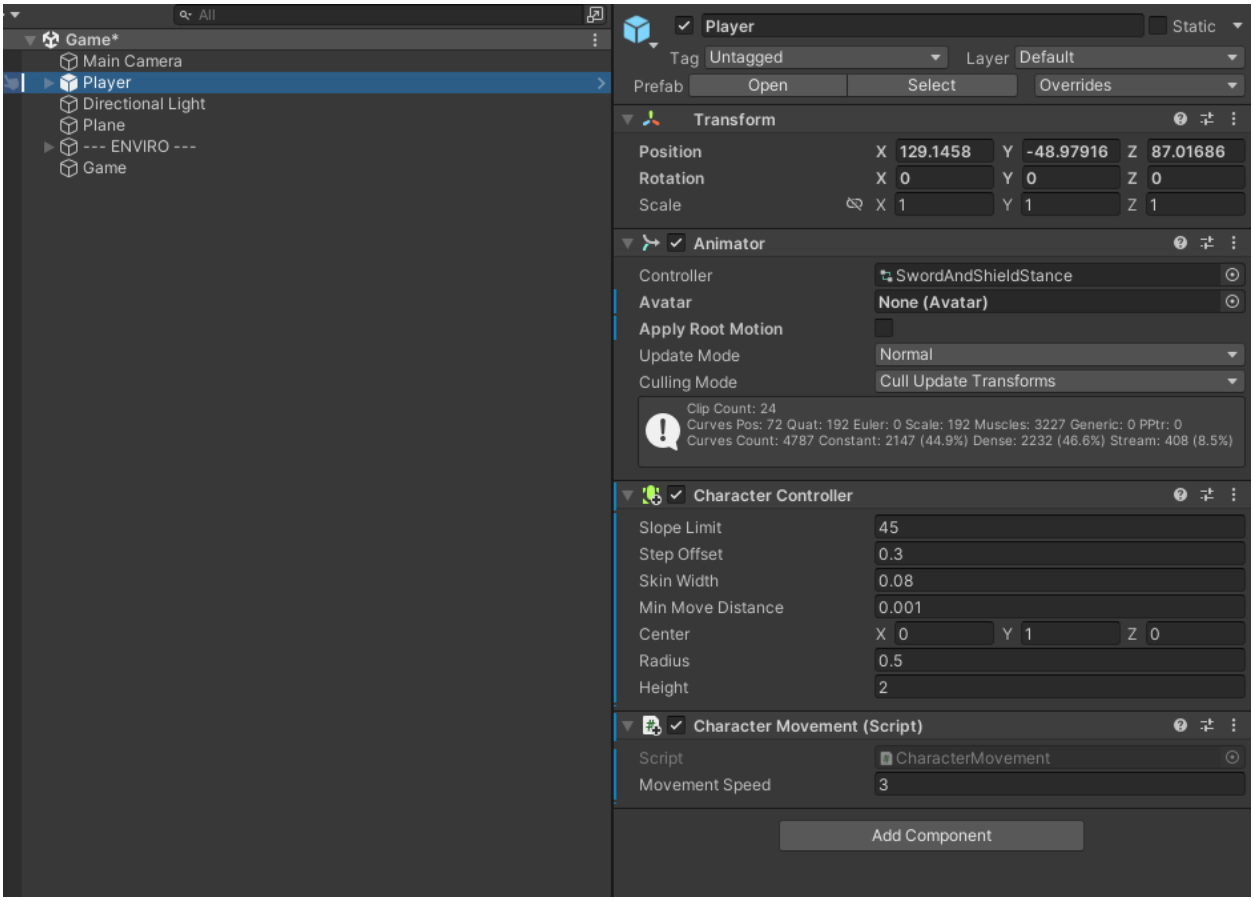
En la primera sesión deberemos en primer lugar importar los distintos packages proporcionados para llevar a cabo la práctica. Entre ellos, se incluye el paquete con los dos personajes.

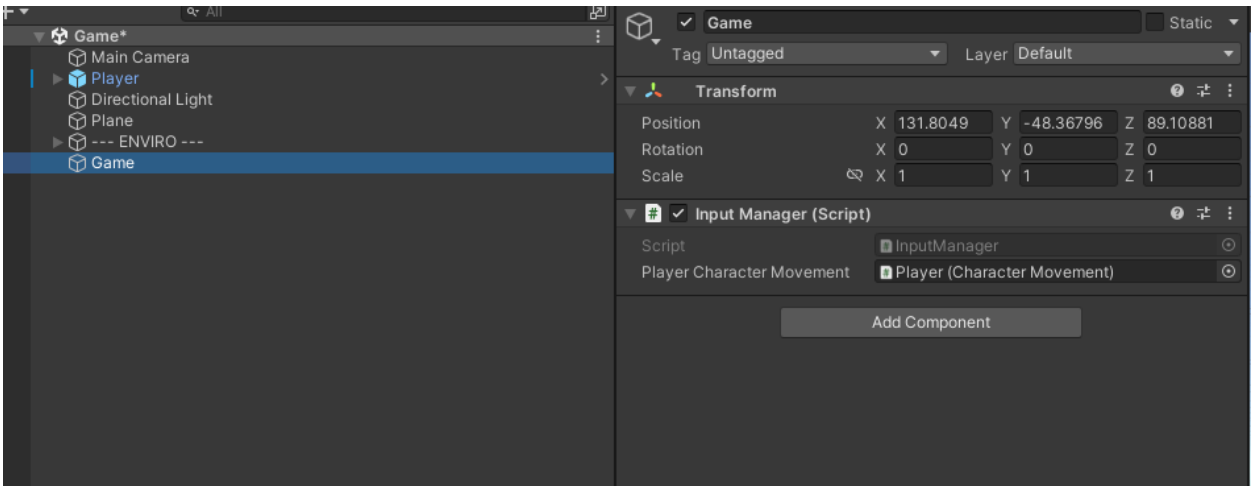
Utilizaremos uno de los prefabs disponibles como base para nuestro personaje. Necesitaremos añadir el CharacterController y ajustar la cápsula. Es importante que no olvidemos eliminar el Animator que viene por defecto para controlar las animaciones y desactivar el Root Motion. Deberíais investigar para que sirven ambas cosas en la documentación de Unity.

Finalmente, el objetivo de la primera sesión será implementar el InputManager, que será un componente del objeto Game. El InputManager enviará la entrada del jugador al CharacterMovement del player, que a su vez hará uso del CharacterController de Unity para mover el personaje.

Se recomienda copiar los templates proporcionados para los scripts y realizar la implementación en base a estos templates.

GameObjects y componentes





1. Input y movimiento

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

public class CharacterMovement : MonoBehaviour
{
    #region paramaters
    /// <summary>
    /// Movement speed of the player. Needs to keep constant while the player moves
    /// </summary>
    /// Desired horizontal movement speed
    [SerializeField]
    private float _movementSpeed = 3.0f;

    /// <summary>
    /// Vertical speed assigned to character when jump starts
    /// </summary>
    [SerializeField]
    private float _jumpSpeed = 20.0f;

    /// <summary>
    /// Minimum vertical speed to limitate falling speed
    /// </summary>
    [SerializeField]
    private float _minSpeed = -10.0f;
    #endregion

    #region references
    /// <summary>
    /// Reference to Player's character controller
    /// </summary>
    private CharacterController _myCharacterController;

    /// <summary>
    /// Reference to Player's Transform
    /// </summary>
    private Transform _myTransform;

    /// <summary>
    /// Reference to Camera's CameraController
    /// </summary>
    private CameraController _cameraController;
    #endregion

    #region properties
    /// <summary>
    /// Horizontal axis input received from InputManager
    /// </summary>
    private float _xAxis;

    /// <summary>
    /// Vertical axis input received from InputManager
    /// </summary>
    private float _zAxis;
```

```
    /// <summary>
    /// Movement direction vector
    /// </summary>
    private Vector3 _movementDirection;

    /// <summary>
    /// Movement vertical speed (needs to be updated every frame due to gravity)
    /// </summary>
    private float _verticalSpeed;
    #endregion

    #region methods
    /// <summary>
    /// Public method to set the horizontal component of input. (Will be called from
InputManager)
    /// </summary>
    /// <param name="x">Received horizontal component</param>
    public void SetHorizontalInput(float x)
    {
        //TODO
    }

    /// <summary>
    /// Public method to set the vertical component of input. (Will be called from
InputManager)
    /// </summary>
    /// <param name="y">Received vertical component</param>
    public void SetVerticalInput(float y)
    {
        //TODO
    }

    /// <summary>
    /// Public method called when the player tries to perform a new Jump. (Will be called
from InputManager)
    /// If the Character is grounded, it overrides current value or _verticalSpeed with
_jumpSpeed.
    /// Otherwise, the request to jump is ignored.
    /// </summary>
    public void Jump()
    {
        //TODO
    }

    #endregion
    /// <summary>
    /// START
    /// Needs to assign _myCharacterController, _myTransform and _cameraController.
    /// If InputManager is already assigned, it will also register the player on it.
    /// </summary>
    void Start()
    {
        //TODO
    }
```

```
    /// <summary>
    /// UPDATE
    /// Needs to calculate and normalize horizontal movement direction
    /// Needs to update vertical speed according to gravity
    /// Finally move the character according to desired _movementSpeed in horizontal and
updated _verticalSpeed
    /// Final details:
    /// -Ensure the character looks in the desired direction according to move direction.
    /// -Ensure to set the vertical following behaviour for camera depending on whether
the character is grounded or not.
    /// </summary>
    void Update()
    {
        //TODO
    }
}
```

2. InputManager

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

public class InputManager : MonoBehaviour
{
    /// <summary>
    /// Reference to Player's CharacterMovement component, to be set from editor first
time,
    /// and reassigned in run time for subsequent times.
    /// </summary>
    [SerializeField]
    private CharacterMovement _playerCharacterMovement;

    #region methods
    /// <summary>
    /// Public method to allow CharacterMovement to register on InputManager so it can
receive input.
    /// </summary>
    /// <param name="playerCharacterMovement">Player's CharacterMovement (Component to be
reigstered)</param>
    public void RegisterPlayer(CharacterMovement playerCharacterMovement)
    {
        //TODO
    }
    #endregion

    /// <summary>
    /// UPDATE
    /// Receive Horizontal input from player, if any, and set it on Player's
CharacterMovement
    /// Receive Vertical input from player, if any, and set it on Player's
CharacterMovement
    /// Receive Jump input from player, if any, and call corresponding method on Player's
CharacterMovement
    /// </summary>
    void Update()
    {
        //TODO
    }
}
```

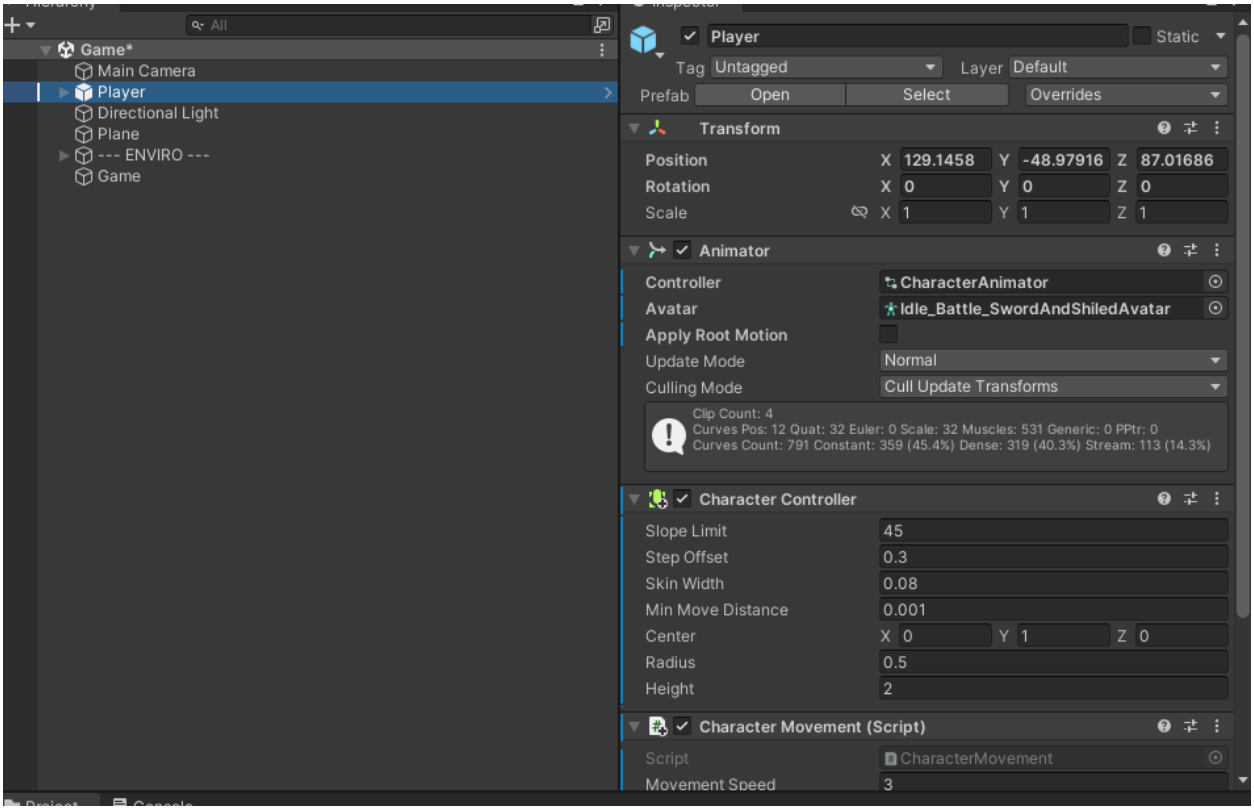
Sesión 2: Animaciones

En la segunda sesión deberemos utilizar algunas de las animaciones que vienen incluidas en el package de personajes.

Antes de empezar debemos saber que para mover a un personaje, Unity necesita tener un Avatar y un AnimatorController.

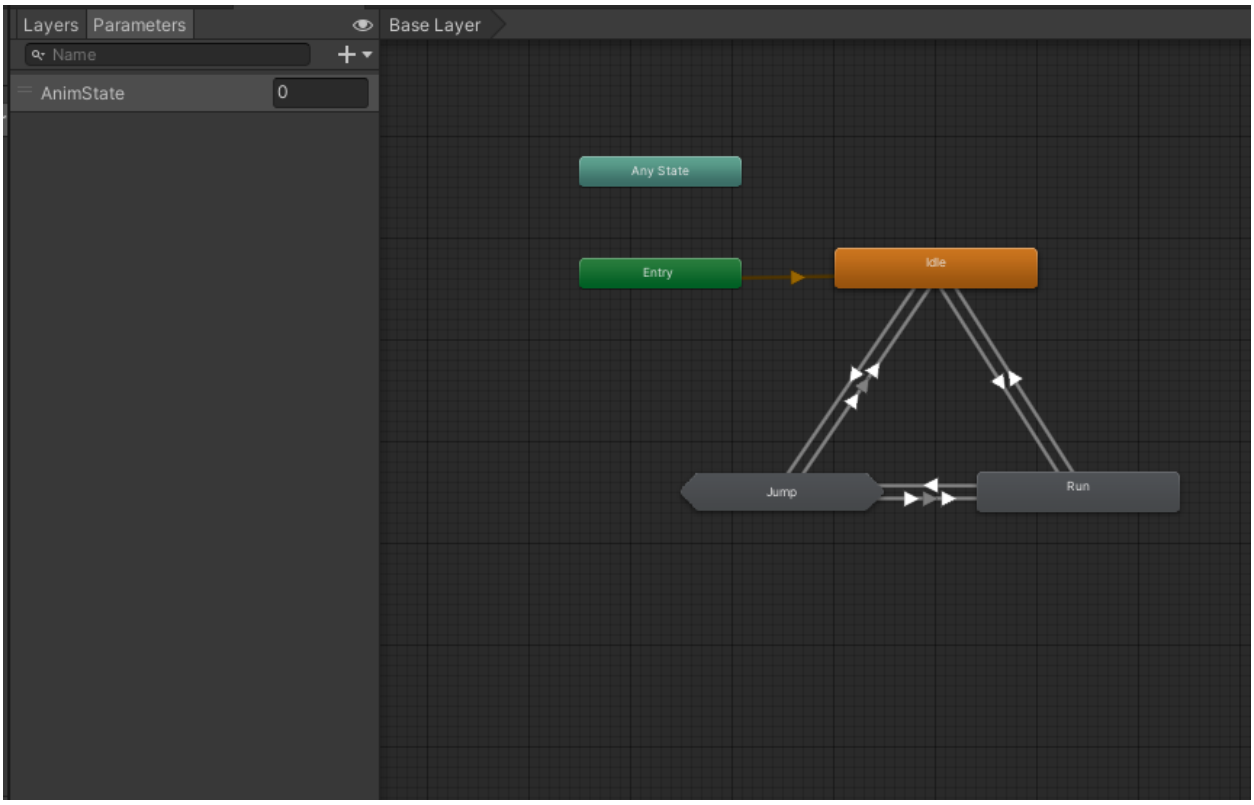
- Para entenderlo de forma sencilla, podeos decir que el avatar es el rig, que contiene la información de la jeraquía de huesos y joints del modelo, sobre los que se aplicarán las animaciones.
- El AnimatorController es una máquina de estados que nos permite definir distintos estados del personaje y las animaciones correspondientes a utilizar en cada caso.

Para asignar ambos, deberemos hacerlo en el animator, como se muestra a continuación:

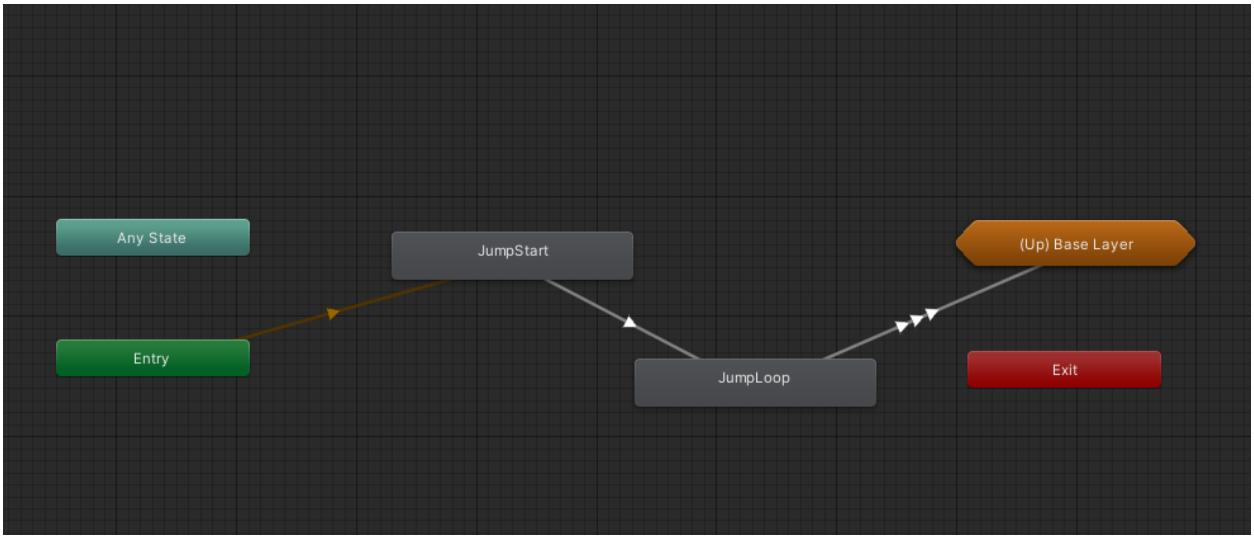


- El avatar asignado viene incluido en el package, sólo hay que asignarlo.

- El animator controller “CharacterAnimator” lo crearemos nosotros, como cualquier otro asset (En este caso, Animator Controller) y lo asignaremos. Haciendo doble click podremos entrar en la ventana para visualizarlo y configurarlo como se ve a continuación:



- Asignamos las animaciones correspondietnes a cada estado, y crearemos el parámetro de tipo entero AnimState, que utilizaremos para regular las transciones entre estados.
- Nota: El estado Jump es en realidad una sub-máquina de estados, que contiene dos estados como se muestra a continuación:



Por último, evaluaremos el estado del CharacterController de Unity para decidir que animación queremos en cada caso, y controlaremos el Animator de Unity mediante nuestro propio componente AnimationComponent.

1. AnimationComponent

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

public class AnimationComponent : MonoBehaviour
{
    #region references
    /// <summary>
    /// Reference to player's Character Controller.
    /// Needs to be assigned on Start
    /// </summary>
    private CharacterController _myCharacterController;

    /// <summary>
    /// Reference to player's Animator.
    /// Needs to be assigned on Start.
    /// </summary>
    private Animator _myAnimator;
    #endregion

    /// <summary>
    /// START
    /// Assign _myCharacterController and _myAnimator
    /// Check if both are correct or disable component
    /// </summary>
    void Start()
    {
        //TODO
    }

    /// <summary>
    /// UPDATE
    /// Evaluate _myCharacterController velocity
    /// Assign the right animation according to this using integer parameter "AnimState"
    /// </summary>
    void Update()
    {
         //TODO
    }
}
```

Sesión 3: Cámara

La posición de la cámara deberá actualizarse en función de la del jugador, siguiéndolo de forma suavizada. Además, deberemos tener en cuenta si el personaje está apoydan en el suelo o no para actualizar la componente vertical de la posición de cámara o no.

1. CameraController

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

public class CameraController : MonoBehaviour
{

    #region parameters
    /// <summary>
    /// Horizonal distance from Camera to CameraTarget.
    /// </summary>
    [SerializeField]
    private float _horizontalOffset = 1.0f;
    /// <summary>
    /// Vertical distance from Camera to CameraTarget.
    /// </summary>
    [SerializeField]
    private float _verticalOffset = 1.0f;
    /// <summary>
    /// Multiplier factor to regulate camera responsiveness to target's movement.
    /// </summary>
    [SerializeField]
    private float _followFactor = 1.0f;
    #endregion

    #region references
    /// <summary>
    /// Camera target transform. Actually, the one the camera needs to follow.
    /// </summary>
    [SerializeField]
    private Transform _targetTransform;
    /// <summary>
    /// Reference to own transform.
    /// </summary>
    private Transform _myTransform;
    #endregion

    #region properties
    /// <summary>
    /// If disabled, the camera does not follow target in vertical axis and keeps its own
Y coordinate.
    /// </summary>
    private bool _yFollowEnabled = true;
```

```
    /// <summary>
    /// Stores own previous position's Y coordinate, to be able to keep it in case
vertical following is disabled.
    /// </summary>
    private float _yPreviousFrameValue;
    #endregion

    #region methods
    /// <summary>
    /// Public methods to allow others to set vertical following behaviour
    /// </summary>
    /// <param name="verticalFollowEnabled"></param>
    public void SetVerticalFollow(bool verticalFollowEnabled)
    {
        //TODO
    }
    #endregion

    /// <summary>
    /// START
    /// Needs to assign _myTransform and initialize _yPreviousFrameValue
    /// </summary>
    void Start()
    {
        //TODO            }

    /// <summary>
    /// LATE UPDATE
    /// Needs to calculate the desired position for the camera.
    /// This calculation will differ depending on _yFollowEnabled.
    /// Once calculated, the new camera position can be assigned accorging to it, in a
smoothed way.
    /// </summary>
    void LateUpdate()
    {
        //TODO
    }
}
```

Sesión 4: Registro de flores y GameManager

Por último, realizaremos una lógica básica de flujo de juego. Las flores se registran en el GameManager al iniciar la partida, y cuando el jugador las ha recogido todas la partida se resetea, volviendo a cargar la escena.

El GameManager tendrá una instancia estática única (singleton) accesible de este modo desde cualquier otro script.

1. GameManager

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

public class GameManager : MonoBehaviour
{
    #region properties
    /// <summary>
    /// Unique allowed instance of GameManager class, self-assigned on Awake (singleton)
    /// </summary>
    static private GameManager _instance;
    /// <summary>
    /// Public accessor so everyone can access the unique instance of the class without
being able to modify it.
    /// </summary>
    static public GameManager Instance
    {
        get { return _instance; }
    }

    /// <summary>
    /// Reference to input manager
    /// </summary>
    private InputManager _input;

    /// <summary>
    /// Public accessor for InputManager so everyone can access it via GameManager
without being able to modify it.
    /// </summary>
    public InputManager Input
    {
        get { return _input; }
    }

    /// <summary>
    /// Current number of registered flowers.
    /// </summary>
    private float _nFlowers;
    #endregion
```

```
    #region methods
    /// <summary>
    /// Public method to allow flowers registration.
    /// </summary>
    public void RegisterFlower()
    {
        //TODO
    }

    /// <summary>
    /// Public method to allow flowers release.
    /// It also needs to check whether all flowers have been released and act
consequentlt if it is the case.
    /// </summary>
    public void ReleaseFlower()
    {
        //TODO                    }

    /// <summary>
    /// In this case, restarting the level means reloading the Game scene.
    /// </summary>
    private void RestartLevel()
    {
        //TODO
    }

    /// <summary>
    /// AWAKE
    /// Needs to check if there already is an assigned _instance of GameManager.
    /// If this is the case, it will destroy its own-object, as this proofs it is not the
first time the scene gets loaded,
    /// and we cannot have two instances of GameManager neither want to have duplicated
InputManagers.
    /// Otherwise, _instance is self-assigned and the object set to not be destroyed on
load.
    /// </summary>
    private void Awake()
    {
        //TODO                    }

    /// <summary>
    /// START
    /// Needs to assign _input
    /// </summary>
    private void Start()
    {
        //TODO
    }
    #endregion
}
```

2. FlowerComponent

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

public class FlowerComponent : MonoBehaviour
{
    /// <summary>
    /// Evaluates if colliding object corresponds to player.
    /// If it does, the Flower is released on GameManager and own object is deactivated.
    /// </summary>
    /// <param name="other">Collider of colliding object</param>
    private void OnTriggerEnter(Collider other)
    {
        //TODO
    }

    /// <summary>
    /// START
    /// Register Flower on GameManager
    /// </summary>
    void Start()
    {
        //TODO
    }
}
```