Video Game Development

Degree in Computer Engineering - 2022/2023

Laboratory Guide nº.2

***Unity* – Physics and the working cycle of a script**

Introduction and Objectives

In this laboratory work, we will deepen the knowledge acquired in the previous exercise. Concepts of collision (*Colliders* , *Collisions* , *Trigger*) and physics (*RigidBody)* will be addressed, as well as some topics related to Unity's internal functioning cycle (*Start*, *Update*) and prefab instantiation.

Theoretical-practical preparation

Collider Component

Colliders are components that allow *Game Objects*, to which they are associated, to react when collisions with other objects occur.
Colliders come in a variety of shapes and sizes, and are represented in the *Scene view by* a green outline. They can have the following primitive forms: sphere, capsule, or box.
For example, when a cube is created, it has, by default, the components shown in Figure 1. The *Box Collider* component that is automatically added is associated with *Unity 's physics engine*.

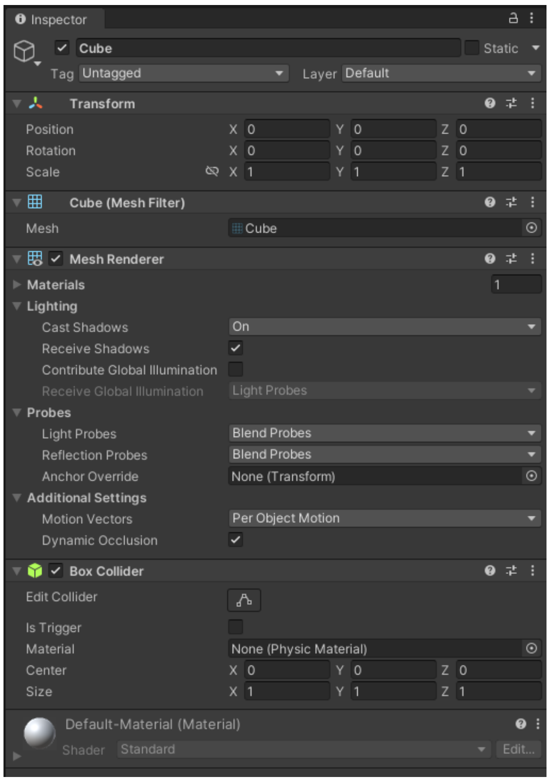
For more complex shapes, we have two options:

1. Combine various primitive shapes by applying *Colliders* to different objects in the hierarchy.

2. Use a *Mesh Collider,* in which one *mesh Collider* has the exact shape of the shape mesh. This last option is the most accurate but has a drawback: *Mesh Collider* takes on the same shape as the model mesh. If the model is very detailed, then *Mesh Collider* will also have a lot of detail, and this can affect performance.

When a collision occurs, several events can be triggered:

- *OnCollisionEnter*: when the collision starts.
- *OnCollisionStay*: while objects are in collision.
- *OnCollisionExit*: when the collision ends.



Inspector

✓ Cube — Static

Tag: Untagged — Layer: Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

Cube (Mesh Filter)

Mesh: Cube

✓ Mesh Renderer

Materials	1
Lighting	
Cast Shadows	On
Receive Shadows	✓
Contribute Global Illumination	
Receive Global Illumination	Light Probes
Probes	
Light Probes	Blend Probes
Reflection Probes	Blend Probes
Anchor Override	None (Transform)
Additional Settings	
Motion Vectors	Per Object Motion
Dynamic Occlusion	✓

✓ Box Collider

Edit Collider			
Is Trigger			
Material	None (Physic Material)		
Center	X 0	Y 0	Z 0
Size	X 1	Y 1	Z 1

Default-Material (Material)

Shader: Standard — Edit...

Figure 1- Base components that are part of a cube

Rigid Body Component

In order to have a “physical behavior” and the collision occurring, it is also necessary to add the *Rigid Body component*. The main purpose of this is to give existence as a physical body to the *Game Object*.

! **Important note:** For a collision to occur at least one of the *Game Objects* must have a *Rigid Body component*.

By selecting the previously created cube, we can add the aforementioned component as shown in Figure 2.

The *rigid body* component provides mass and gravity, which can be configured by changing the properties seen in Figure 3.

Summary of each property:

- ***Mass*** – Value in kilograms that will mainly influence the bounces between objects when they collide.

- ***Drag*** – The attrition of this body.

- ***Angular Drag*** – The attrition for angular forces.

- ***Use gravity*** – Allows defining whether we want to automatically apply a force corresponding to gravity to the object.

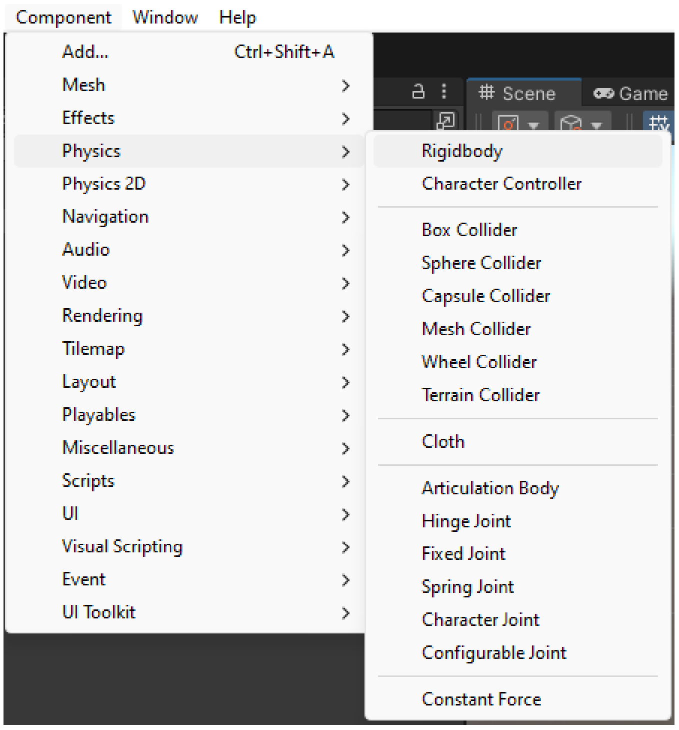
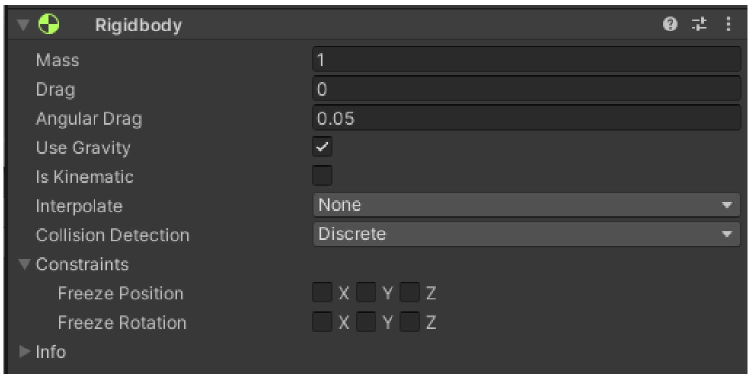


Figure 2– Creating a Rigid Body component



Rigidbody	
Mass	1
Drag	0
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	☐
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	☐ X ☐ Y ☐ Z
Freeze Rotation	☐ X ☐ Y ☐ Z
Info	

Figure 3- Rigid Body Component

- ***Is Kinematic*** – An object that is set to kinematic is fixed in the scene and is not subject to external forces. It will influence other objects, but its positioning will remain intact. It is useful for parts of the scenery that are unalterable.

- ***Interpolate*** – Alternative methods to calculate and/or predict physics in between the discreet calculations.

- ***Collision Detection*** – Alternative methods to detect the collisions.

- ***Constraints*** – Allows you to restrict the physical behavior (translations and rotations independently) on a given axis. For example, in a platform game, we might want to limit the character's movement on the screen plane.

Triggers

Collider component can be a *Trigger*. When the component is a *Trigger*, the associated *Game Object* will not participate in collisions, instead objects will pass through it and the following events can be fired:

- *OnTriggerEnter*: the object entered the *Trigger*.
- *OnTriggerStay*: the object is inside the *Trigger*.
- *OnTriggersExit*: the object exited on *Trigger*.

Triggers are useful when we want *to* detect the interaction between two objects without a visible collision. For example, if we want to detect whether a *player* has passed through a door, we place a *Trigger* at the location of that door. When the *player* passes through the *Trigger* (there won't be a collision as it passes through it), the event can be handled.

! **Important Note:** As with collisions, at least one of the objects must have a *Rigid component Body*.

As a rule, *Triggers* are objects to which no physical forces are applied and which remain static, with the object that collides with the *Trigger* having the *Rigid Body* component.

Adding Physical Forces

For a *game Object*, having a *Rigid Body*, to move according to a given force, we can use the *addForce method* applied to the *Rigid Body*. This method receives the force to be applied as a parameter.
In the following *script*, the cursor arrows (using `Input.GetAxis (...))` on the keyboard are used to guide the object in a plane:

```
private Rigid body rb ;

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

...

void FixedUpdate () {
    float moveHorizontal = Input . GetAxis ( "Horizontal" );
    float moveVertical = Input . GetAxis ( "Vertical" );

    Vector3 movement = new Vector3 ( moveHorizontal , 0 , moveVertical );
    rb . AddForce ( movement * speed );
}
```

Example – Collisions and Triggers

In the *Scene* of Figure 4, there is a cube with the *Box Collider* component, created by default, and a sphere on top of the cube with a *Sphere component Collider,* which is also created by default. Play this *Scene* and press *Play*. What happens?

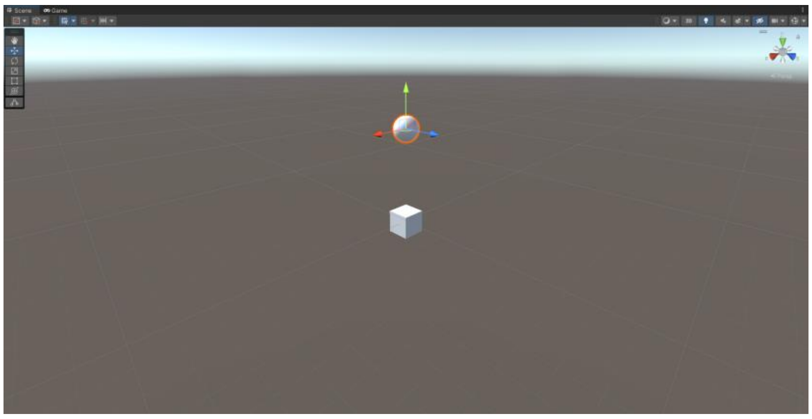


Figure 4- Scene with a cube and a sphere aligned vertically.

What if we add a *Rigid Body component* to the sphere, as in Figure 5?

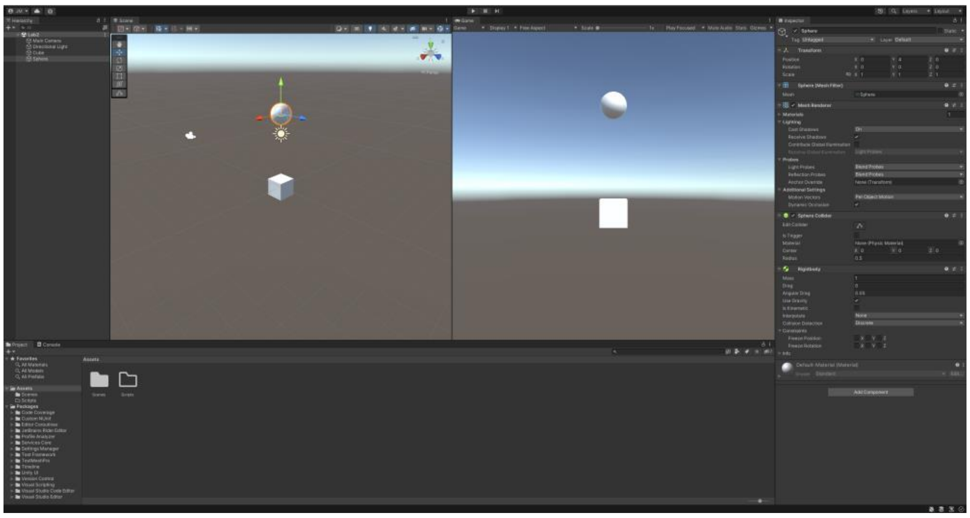


Figure 5- Scene with a cube and a sphere (with *Rigid component Body*) aligned vertically.

Try moving the sphere a little to the right so that, when it falls on top of the cube, it hits the edge and does not rest on top (that is, so that the collision ends), as in Figure 6.

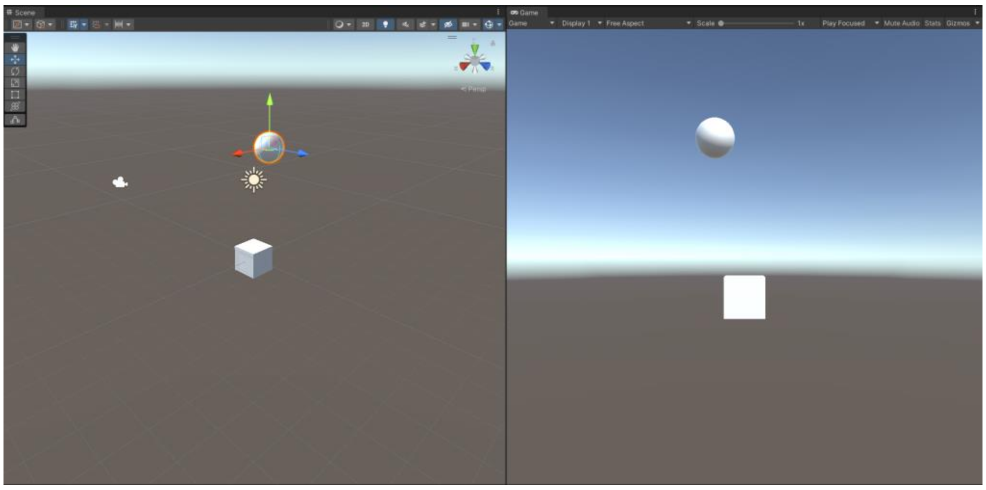


Figure 6- Scene with a cube and sphere aligned vertically, where the sphere has a Rigid Body component.

Now, add the following *script* to the sphere:

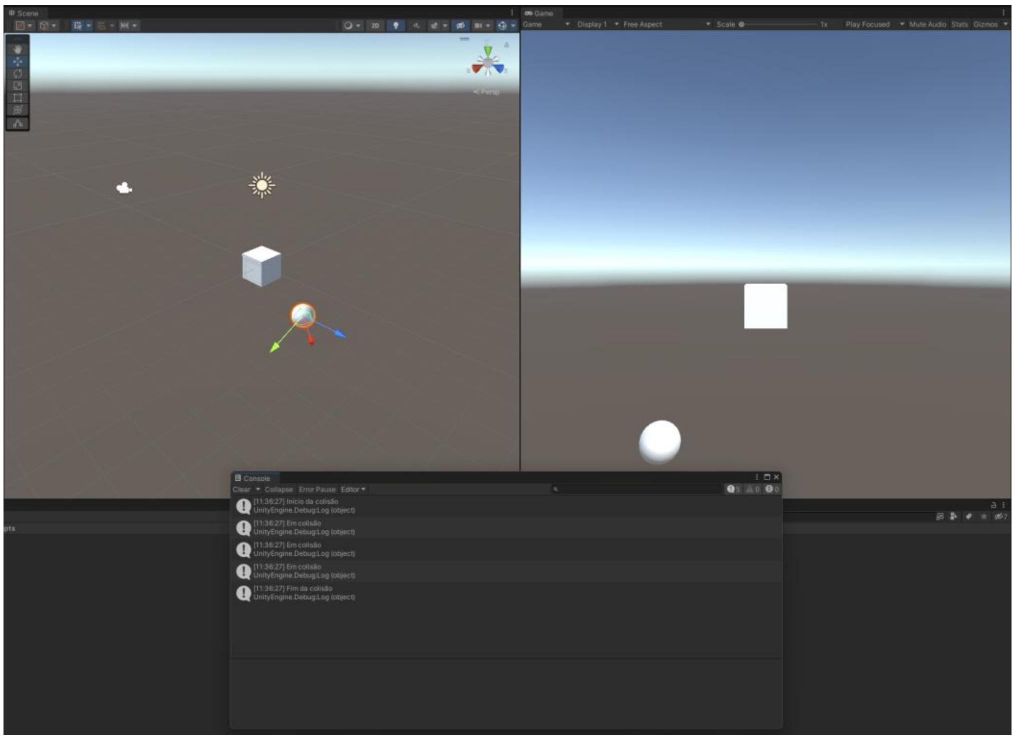
```
public class SphereController : MonoBehaviour {

        void OnCollisionEnter ()
        {
                Debug .Log ( "Collision start" );
        }

        void OnCollisionStay ()
        {
                Debug .Log ( "On Collision" );
        }

        void OnCollisionExit ()
        {
                Debug .Log ( "End of collision" );
        }
}
```

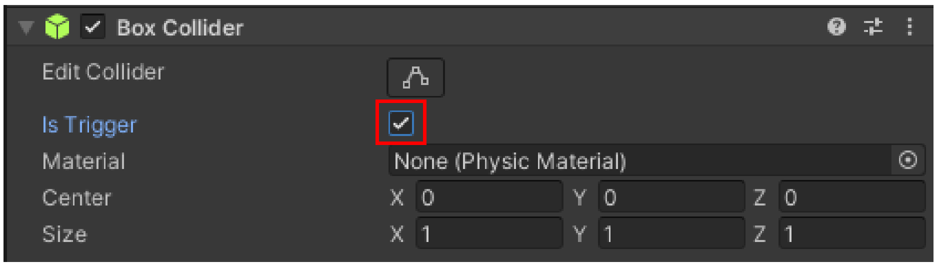
and watch the console messages, as in Figure 7.



```
[11:36:27] Início da colisão
UnityEngine.Debug:Log (object)
[11:36:27] Em colisão
UnityEngine.Debug:Log (object)
[11:36:27] Em colisão
UnityEngine.Debug:Log (object)
[11:36:27] Em colisão
UnityEngine.Debug:Log (object)
[11:36:27] Fim da colisão
UnityEngine.Debug:Log (object)
```

Figure 7– Console Messages (Collider) .

Now define the cube as a *Trigger*, as in Figure 8.



Box Collider
Edit Collider
Is Trigger ✓
Material None (Physic Material)
Center X 0 Y 0 Z 0
Size X 1 Y 1 Z 1

Figure 8– Cube that is a Trigger .

Add the following *script* to the cube:

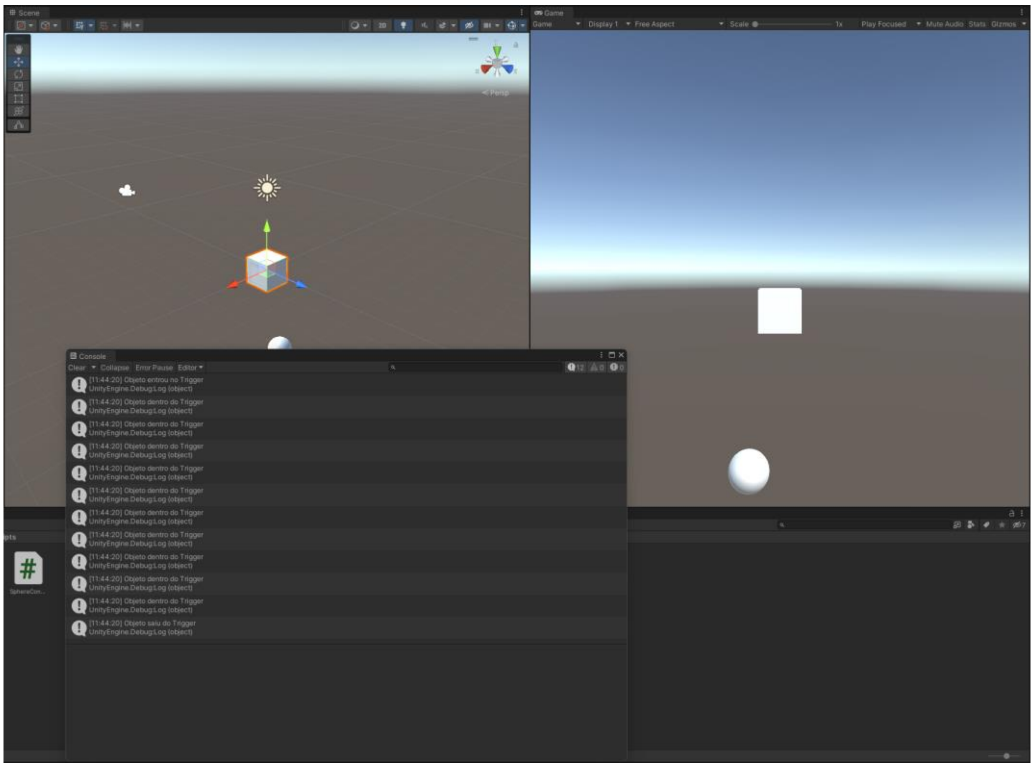
```
public class CubeController : MonoBehaviour {

        void OnTriggerEnter ()
        {
                Debug .Log ( "Object entered Trigger " );
        }

        void OnTriggerStay ()
        {
                Debug.Log ( " Object inside Trigger " );
        }

        void OnTriggerExit ()
        {
                Debug.Log ( "Object exited Trigger " )
        ; }
}
```

And watch the console messages, as presented in Figure 9.



```
[11:44:20] Objeto entrou no Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto dentro do Trigger
UnityEngine.Debug:Log (object)
[11:44:20] Objeto saiu do Trigger
UnityEngine.Debug:Log (object)
```

Figure 9– Console Messages (Trigger) .

Inner working cycle of a script in Unity

When executed, a Unity script triggers several *event functions* in a predetermined order. Figure 10 summarizes how Unity sorts and repeats these *event functions* in the lifetime of a script.

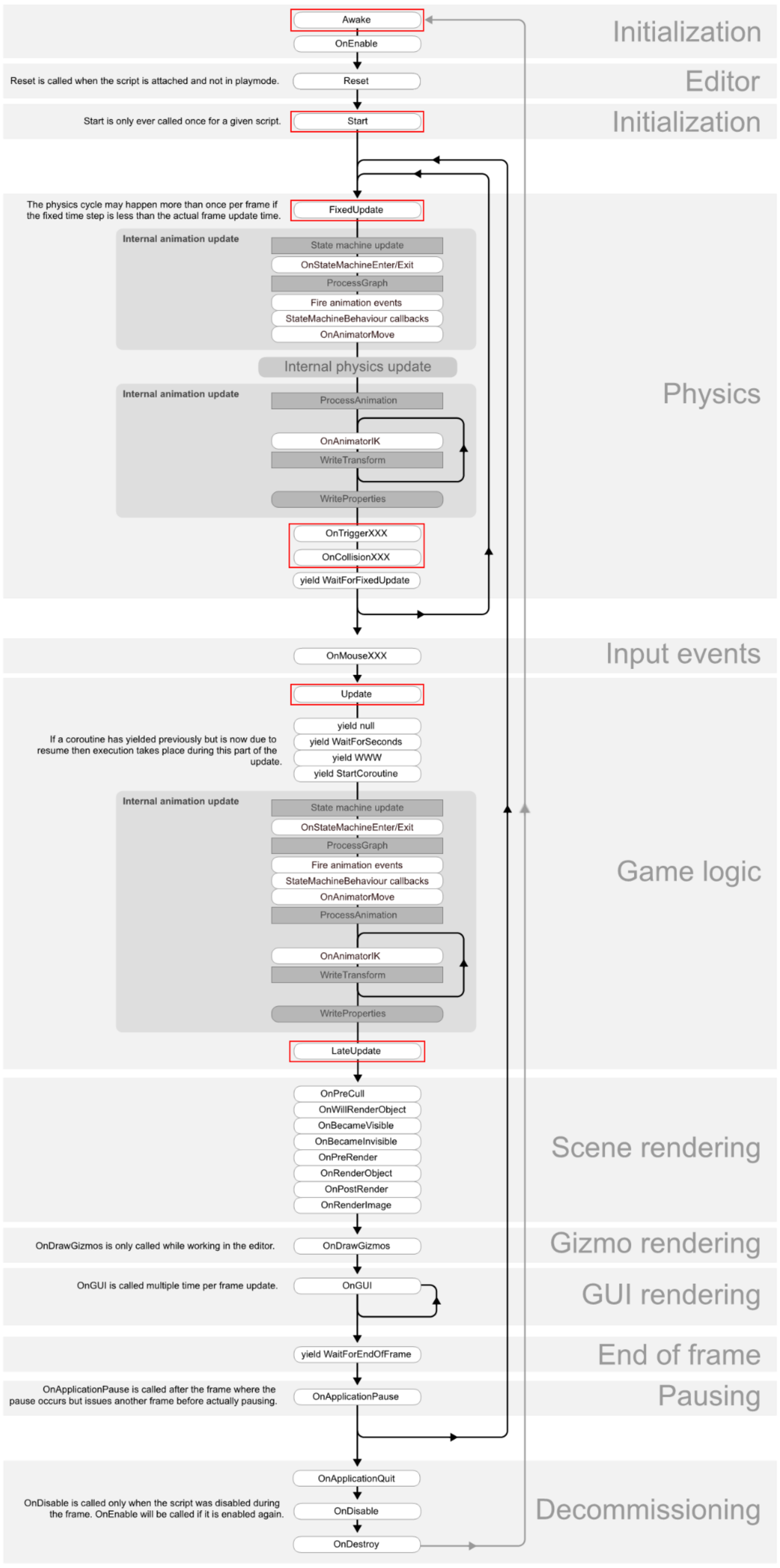

Initialization
Awake
OnEnable
Editor
Reset is called when the script is attached and not in playmode.
Reset
Initialization
Start is only ever called once for a given script.
Start
Physics
The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.
FixedUpdate
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
Internal physics update
Internal animation update
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
OnTriggerXXX
OnCollisionXXX
yield WaitForFixedUpdate
Input events
OnMouseXXX
Game logic
Update
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.
yield null
yield WaitForSeconds
yield WWW
yield StartCoroutine
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
LateUpdate
Scene rendering
OnPreCull
OnWillRenderObject
OnBecameVisible
OnBecameInvisible
OnPreRender
OnRenderObject
OnPostRender
OnRenderImage
Gizmo rendering
OnDrawGizmos is only called while working in the editor.
OnDrawGizmos
GUI rendering
OnGUI is called multiple time per frame update.
OnGUI
End of frame
yield WaitForEndOfFrame
Pausing
OnApplicationPause is called after the frame where the pause occurs but issues another frame before actually pausing.
OnApplicationPause
Decommissioning
OnApplicationQuit
OnDisable is called only when the script was disabled during the frame. OnEnable will be called if it is enabled again.
OnDisable
OnDestroy

Figure 10– Operating cycle of a script in Unity .

Note the functions that are highlighted in red (*Awake* , *Start* , *FixedUpdate* , *Update* , *LateUpdate*) and their moment of execution in the cycle.

Awake function

This function is always called before any *Start function* and right after the instantiation of a prefab (see topic Instantiation). If a *GameObject* is inactive during initialization, the *Awake function* will not be called until it is activated.

It is a useful function for initializing certain types of variables that have to change values before calling the *Start function*.

Start function

The *Start function is called before the first frame* is updated, only if the script instance is active (*enabled*).
For objects that are part of the scene, the *Start function* is called in all scripts before the *Update function.*

Typically, it is in this function where we initialize the variables. The *Start function* will be called only once for each script where it is defined.

Update functions (Fixed Update, LateUpdate)

During the game, it is natural to want to monitor and execute certain functionalities related to the game's logic, the various interactions, animations, camera positioning, etc. There are several events that can be used for this, the most common of which is to execute tasks within the *Update function.* But there are also other extremely useful functions.

FixedUpdate: This function is generally called more frequently than *Update.* It may be called several times per *frame* if the *frame rate* is low and may not be called at all between *frames* if the *frame rate* is high.
All physics related calculations and updates occur immediately after *FixedUpdate.* When applying motion calculations within *FixedUpdate*, it is not necessary to multiply values by Time.deltaTime. This happens because *FixedUpdate* is called with a reliable *timer*, regardless of the *frame rate*.

Update : The *Update function* is called once per *frame* . It is the main function for *frame updates*.

LateUpdate: LateUpdate is a function called once per frame, after the Update function has finished. All calculations performed on Update will complete when LateUpdate starts.
A common use for LateUpdate would be a third-person camera. If the game character moves and turns within Update, all camera movement and rotation calculations can be performed in LateUpdate. This will ensure that the character has moved completely before the camera tracks its position.

! **Important Note:** The order in which an event function is called for different instances of the same *MonoBehaviour subclass* cannot be specified. For example, the *Update function* of a *MonoBehaviour* can be called before or after the *Update* function for the same *MonoBehaviour* in another *GameObject* — including its own *GameObject* (parent or child).

Instantiation

The objective of *Instantiate* is to clone a given object and return the clone. This function makes a copy of an object like the "Duplicate" command in the editor. If a clone of a *GameObject* is being made, a position and rotation can be specified (otherwise the default is the position and rotation of the original *GameObject).* If a component is being cloned, the *GameObject* it is attached to will also be cloned, again with an optional position and rotation.

When cloning a *GameObject* or *Component*, all child objects and components will also be cloned with their properties set to those of the original object.

Instantiate function can be used to create new objects on the fly. Examples include objects used for projectiles or particle systems for explosion effects.

```
using UnityEngine ;

// Instantiate a rigidbody then set the velocity
public class Example : MonoBehaviour
{
    // Assign a Rigidbody component in the inspector to instantiate
    public Rigidbody projectile ;

    void Update ()
    {
        // Ctrl was pressed, launch a projectile
        if ( Input.GetButtonDown ( "Fire1" ))
        {
            // Instantiate the projectile at the position and rotation of this transform
            Rigidbody clone ;
            clone = Instantiate (projectile ,transform.position , transform.rotation );

            // Give the cloned object an initial velocity along the current
            // object's Z axis
            clone.velocity = transform.TransformDirection (Vector3.forward * 10 );
        }
    }
}
```

Prefabs

Prefabs are *GameObjects* that can be configured and saved with all their components and values in a reusable way. An *asset* is created that works as a *template* and with which new instances can be generated in the scene.

Whenever you think of reusing a certain *GameObject* (a specific part of the scenery like a tree, for example) in different areas of the scene (or in multiple scenes), that *GameObject* should be converted into a *prefab*. Any changes that are made to the *prefab* will be reflected in all instances of that *prefab*.
It is also possible to make changes to specific instances of *prefabs* allowing for slight differences between these types of *assets* and allowing for a more personalized configuration.

Prefabs must be used when we intend to instantiate certain *GameObjects* at runtime, *objects* that *did not* initially exist in the scene.

Laboratory work - Exercise

In this exercise, a scene should be created with a character who moves in a certain area and who has the possibility of shooting bullets to eliminate his enemies.

! **Important note:** The scene does not necessarily have to be the same or even use the same *assets.* We count on your originality.

It is, however, necessary to consider that:

- The camera must follow the player (in the image represented by a capsule) in a 3rd person perspective.
- The player must move with the keys commonly used for character movement (WASD and direction arrows). Lateral movements must be accompanied by rotation. The CharacterController component should also be used to assist in this movement.
- Enemies should be implemented by creating *prefabs* placed directly in the scene and, if desired, each can have a different value for *hitpoints*. The player must be "surrounded" by enemies, but they don't have to move.
- Shots must be fired using the *space key*, each of which must be implemented using a *prefab* that must be instantiated at *runtime* after the fire key is pressed.
- Physical forces must be used for the movement of the bullets, being possible to adjust the force value through the *inspector*.
- Each shot must collide with enemies and remove a value of 10 from the *hitpoints* they currently have. If this value is equal to or less than 0, the enemy must be destroyed and disappear from the scene.
- Use the "Enemy" tag for enemies to indicate that collisions should only be performed with that type of GameObject .

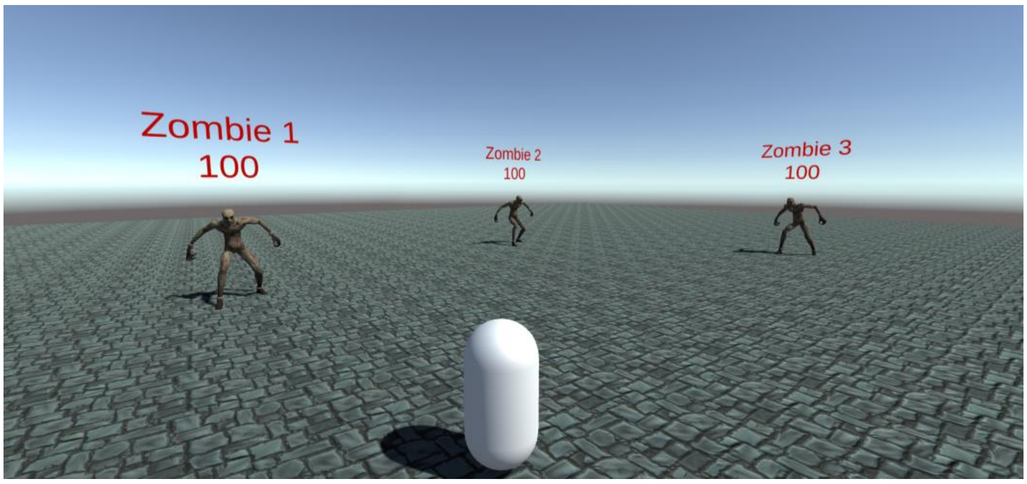


Figure 11 - Intended final result of this work.

Extra Features

The name of each enemy, as well as the respective *hitpoints*, should be considered as an extra for this laboratory work. Each of the enemies must have a unique name associated with it and the value of the referred *hitpoints* must be independent, so that it is possible to verify the value that each one still has available.

In the case of using bullets in a game, it is important that they are eliminated in case they do not collide with any element present in the scene (enemies, walls, or other objects). Thus, you should implement a mechanism to eliminate them after 3 seconds.

Complementary material for consolidation

Support Tutorials – Complementary for consolidation

colliders
https://learn.unity.com/tutorial/3d-physics#5c7f8528edbc2a002053b50d

Colliders as Triggers
https://learn.unity.com/tutorial/3d-physics#5c7f8528edbc2a002053b50e

Rigid Bodies
https://learn.unity.com/tutorial/3d-physics#5c7f8528edbc2a002053b50f

Adding Physics Forces
https://learn.unity.com/tutorial/3d-physics#5c7f8528edbc2a002053b510

Detecting Collisions (On Collision Enter)
https://learn.unity.com/tutorial/3d-physics#5c7f8528edbc2a002053b515

Official Documentation

Fixed update
https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html
https://unity3d.com/en/learn/tutorials/topics/scripting/update-and-fixedupdate

Tags
https://docs.unity3d.com/Manual/Tags.html

collider
https://docs.unity3d.com/ScriptReference/Collider.html

rigid body
https://docs.unity3d.com/ScriptReference/Rigidbody.html

Inner working cycle of a script in Unity
https://docs.unity3d.com/Manual/ExecutionOrder.html

Instantiate
https://docs.unity3d.com/ScriptReference/Object.Instantiate.html

prefabs
https://docs.unity3d.com/Manual/Prefabs.html