QuickBaseGameManager

The *QuickVR* library offers a tool to easily create and manipulate the workflow of our application. The *QuickBaseGameManager* is basically the main of our application, the entry point. It subdivides the logic of the application into 3 main blocks:

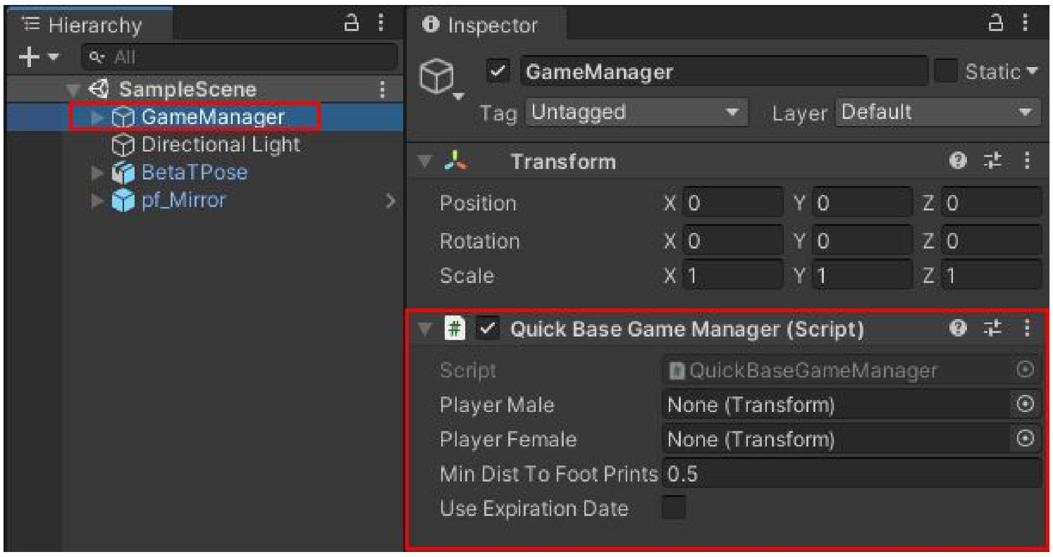
The *StagesPre* contains the logic that has to be executed when the application starts, and before the main logic of the application. Here we typically want to calibrate the existing VR devices or make sure that the HMD is correctly adjusted so the user will see the scene clearly.

The *StagesMain* contains the main logic of the scene, what the application has to do.

Finally, the *StagesPost* contains the logic that we want to execute just before closing the application. For example, here we want to save some data to the disk or to the cloud that we have acquired during the execution of the application.

Also there is an event triggered before and after the execution of these 3 main blocks, so for example, if we want to do an action just after the Stages Pre are finished, but before the Stages Main, we can do it by registering to the corresponding event.

So let's try it out. Create a new *GameObject* on your scene and name it *GameManager.* Add the component *QuickBaseGameManager.* Press play, put on your HMD and follow the instructions appearing on the screen. Once the calibration process is finished, you will appear on the scene, but this time you are already calibrated.



GameManager
Tag Untagged Layer Default

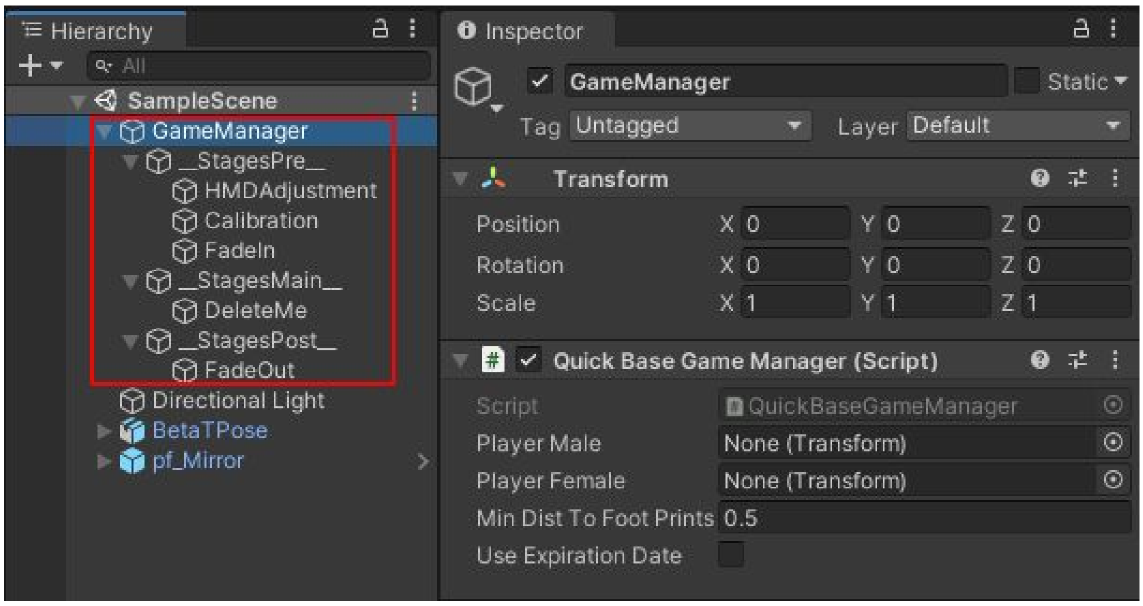
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Quick Base Game Manager (Script)
Script QuickBaseGameManager
Player Male None (Transform)
Player Female None (Transform)
Min Dist To Foot Prints 0.5
Use Expiration Date

QuickStages

We have seen previously that the logic of an application is subdivided into 3 main blocks: *StagesPre, StagesMain* and *StagesPost.* But what is exactly a *Stage?* A *Stage* (named *QuickStage* following the *QuickVR* naming convention) can be seen as each one of the steps into we can subdivide the logic of our application. Each one of this step can be arbitrary complex and is the decision and responsibility of the programmer to subdivide the logic into the appropriate number of *QuickStages.*

So let's see a practical example. If you expand the *GameManager* object you will see that the following structure has been automatically created:



The GameObject *__StagesPre__* is composed by the following stages:

- **HMDAdjustment:** Some text is displayed in order to verify that the HMD is correctly adjusted.
- **Calibration:** The VR devices are calibrated taking the current direction that the participant is looking at as the forward vector reference.
- **FadeIn:** The scene is faded-in gradually.

Next the stages on *__StagesMain__* are executed. By default there is only one stage:

- **DeleteMe:** This is a dummy stage with no specific logic and that never ends. As the name indicates, you want to delete this *GameObject* and replace it with the *QuickStages* defining the main logic of your application.

Finally, the stages on *__StagesPost__* are executed when the game is going to close. This happens when you press the *Esc* key or you manually call the *Finish* function in *QuickBaseGameManager.* By default it contains a single stage:

- **FadeOut:** The scene is gradually faded-out.

The *QuickBaseGameManager* and *QuickStages* approach offers us a highly customizable workflow:

- We can enable or disable stages by simply enabling or disabling the *GameObject* containing this stage.
- We can change the order of execution of a stage by simply moving the *GameObject* on the *Hierarchy* at the corresponding desired position.
- There are many stages already defined, like the ones used for the calibration, producing a fade of the screen, saving data, but also other types of stages that allows us to emulate an *if* or a *loop*.

You can easily create your own *QuickStages* by inheriting from an already defined *QuickStage* if you have to customize its behavior, or from *QuickStageBase.* In fact, all the stages inherit at some point from *QuickStageBase.*

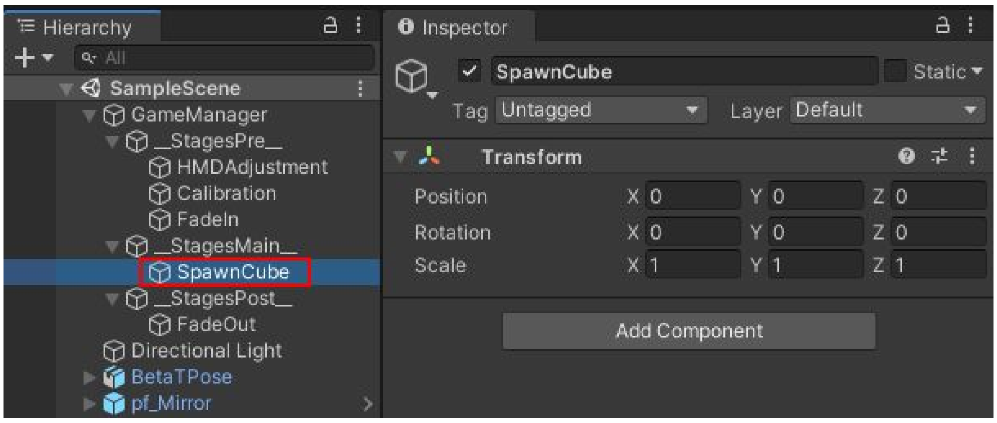
There are 3 main functions that we have to take into consideration when creating a new *QuickStage*:

- **Init:** This function is called when the stage starts its execution. You want to set here any logic that is executed one time, when the stage starts.
- **CoUpdate:** This is the main coroutine that you want to override and here is where you usually want to set the logic of this specific stage. When *CoUpdate* ends, the next function *Finish* is automatically called. As *QuickStageBase* inherits from *MonoBehaviour,* you can also control the logic of the stage in the *Update, LateUpdate, FixedUpdate…*But in this case, you are responsible of calling the function *Finish* when the logic of this stage is over, otherwise the control flow will remain on this stage.
- **Finish:** This function is called when the stage finishes. This process is automatically done when *CoUpdate* ends, or called manually as explained previously. It basically ends the current stage and passes the flow control to the next stage in the hierarchy.

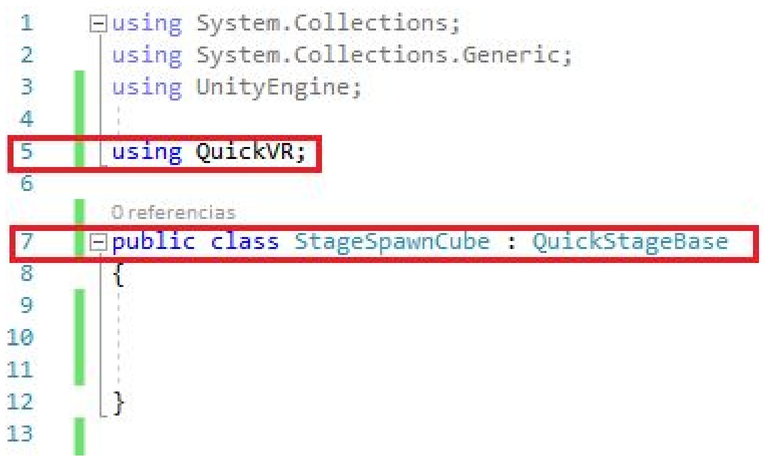
Creating a new QuickStage

Let's see a step by step example by adding something to do on *__StagesMain__*. Imagine that the logic of our application is as follows: *"Wait for 3 seconds and then spawn a cube"*.

So first of all, replace the *DeleteMe GameObject* in *__StagesMain__* by a new *GameObject* and name it *SpawnCube.*

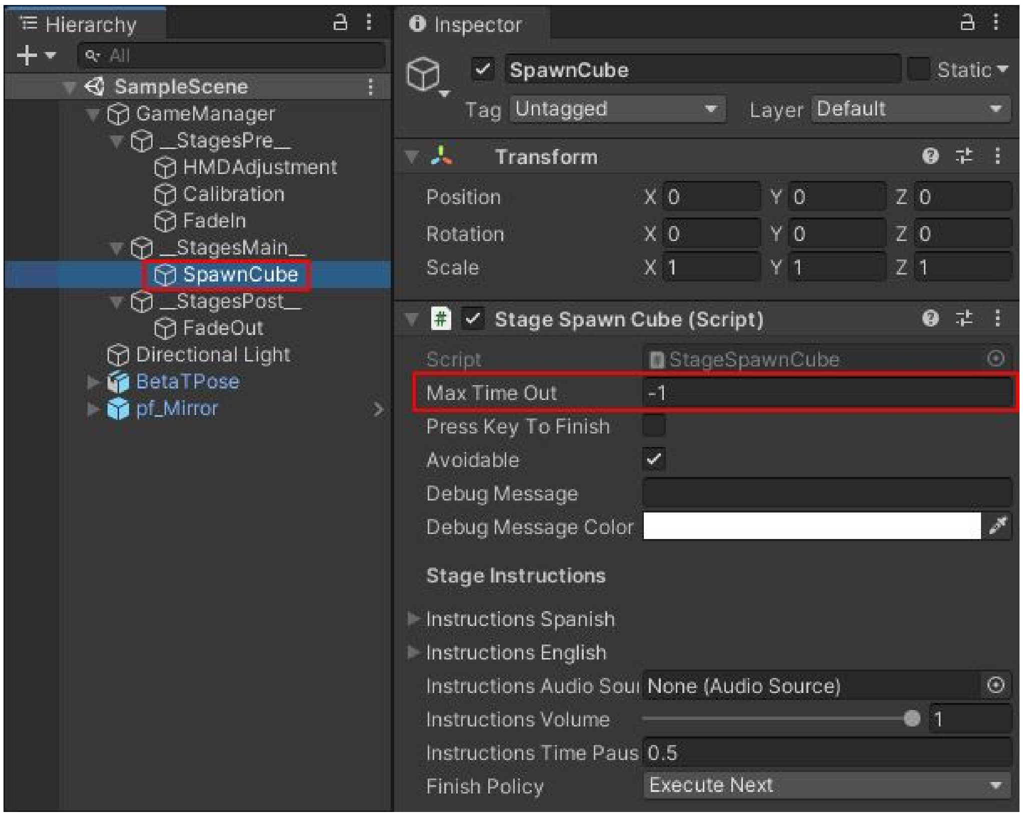


Now add a new script component and name it *StageSpawnCube.* This will be a subclass of *QuickStageBase.*



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

using QuickVR;

0 referencias
public class StageSpawnCube : QuickStageBase
{


}
```

Set the attribute *MaxTimeOut* of this stage to *-1.* This will produce this stage to never end until we press *Esc.* At this point, your hierarchy should look like this:



Hierarchy

SampleScene
- GameManager
  - __StagesPre__
    - HMDAdjustment
    - Calibration
    - FadeIn
  - __StagesMain__
    - SpawnCube
  - __StagesPost__
    - FadeOut
- Directional Light
- BetaTPose
- pf_Mirror

Inspector

SpawnCube — Static
Tag Untagged — Layer Default

Transform

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

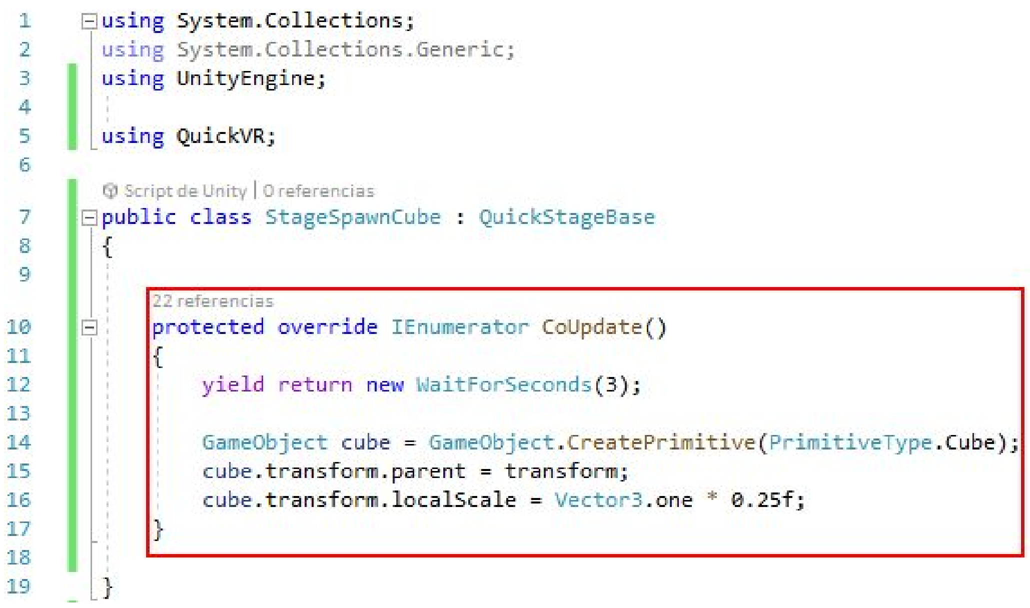
Stage Spawn Cube (Script)

Script	StageSpawnCube
Max Time Out	-1
Press Key To Finish	
Avoidable	✓
Debug Message	
Debug Message Color	

Stage Instructions

Instructions Spanish	
Instructions English	
Instructions Audio Sour	None (Audio Source)
Instructions Volume	1
Instructions Time Paus	0.5
Finish Policy	Execute Next

Now let's implement the logic of this stage. We want to *"wait for 3 seconds and then spawn a cube"*. So override the coroutine *CoUpdate* of *QuickStageBase* as follows.



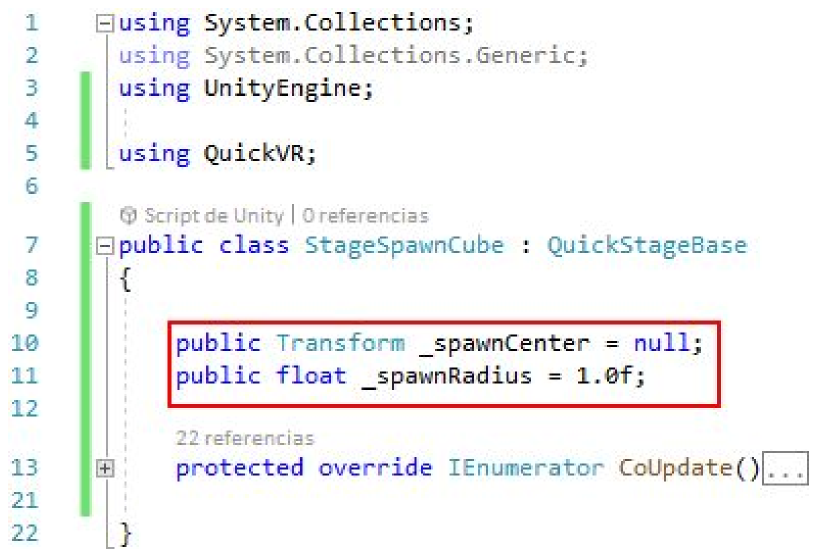
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using QuickVR;

Script de Unity | 0 referencias
public class StageSpawnCube : QuickStageBase
{

    22 referencias
    protected override IEnumerator CoUpdate()
    {
        yield return new WaitForSeconds(3);

        GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
        cube.transform.parent = transform;
        cube.transform.localScale = Vector3.one * 0.25f;
    }

}
```

Press play and you will see that the logic is executed correctly and a cube is spawned, after waiting for 3 seconds.

Creating a Loop using QuickStages

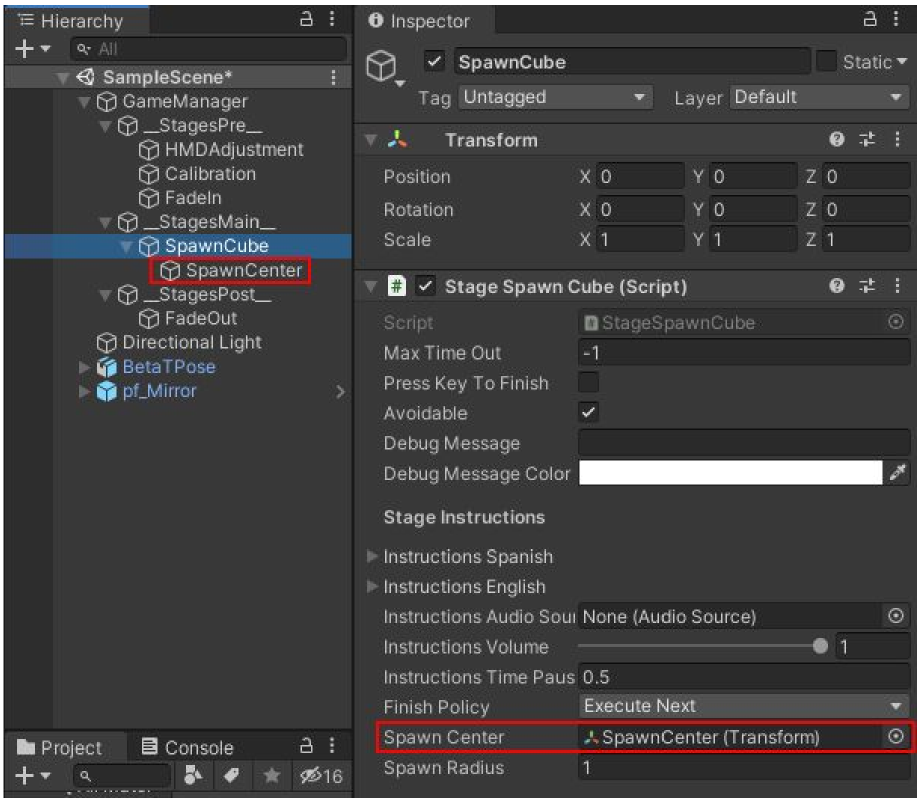
Now suppose that you want to repeat this stage 5 times. Of course you could do a *for loop* in order to spawn 5 cubes, but let's do it using stages, just to show the power of what can you do with this.

First of all, create two new public variables, one called *_spawnCenter* which will be a *Transform,* and another one called *_spawnRadius* of *float* type.



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

using QuickVR;

Script de Unity | 0 referencias
public class StageSpawnCube : QuickStageBase
{

    public Transform _spawnCenter = null;
    public float _spawnRadius = 1.0f;

    22 referencias
    protected override IEnumerator CoUpdate()...
}
```

Back on the scene hierarchy, create a new child *GameObject* of *SpawnCube.* Name it *SpawnCenter* and drag it to the new *Spawn Center* field that has appeared in the component *StageSpawnCube.*



Hierarchy
- SampleScene*
  - GameManager
    - __StagesPre__
      - HMDAdjustment
      - Calibration
      - FadeIn
    - __StagesMain__
      - SpawnCube
        - SpawnCenter
    - __StagesPost__
      - FadeOut
  - Directional Light
  - BetaTPose
  - pf_Mirror

Inspector

SpawnCube — Static
Tag Untagged — Layer Default

Transform

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

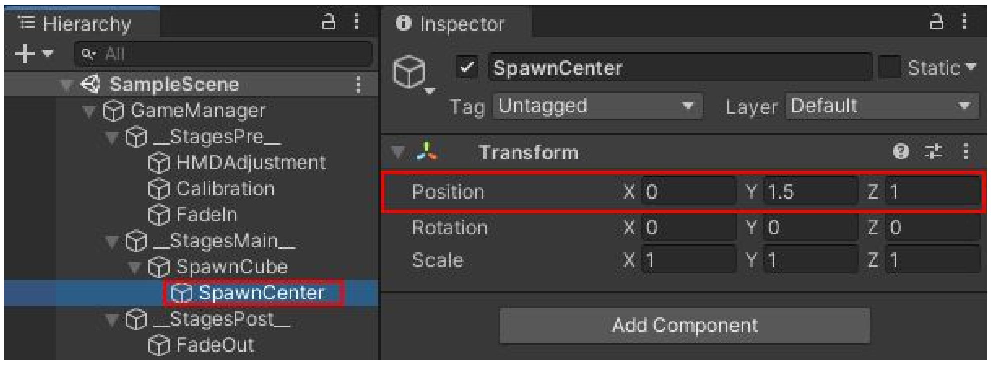
Stage Spawn Cube (Script)

Script	StageSpawnCube
Max Time Out	-1
Press Key To Finish	
Avoidable	✓
Debug Message	
Debug Message Color	

Stage Instructions

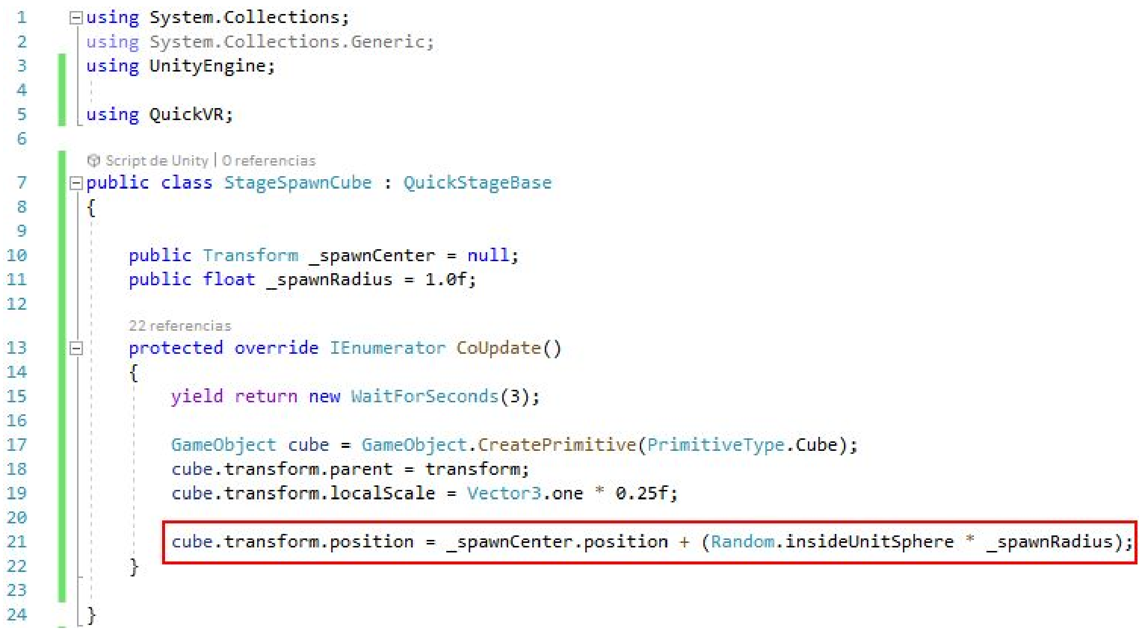
Instructions Spanish	
Instructions English	
Instructions Audio Sour	None (Audio Source)
Instructions Volume	1
Instructions Time Paus	0.5
Finish Policy	Execute Next
Spawn Center	SpawnCenter (Transform)
Spawn Radius	1

Move the *SpawnCenter* object to (0, 1.5, 1).



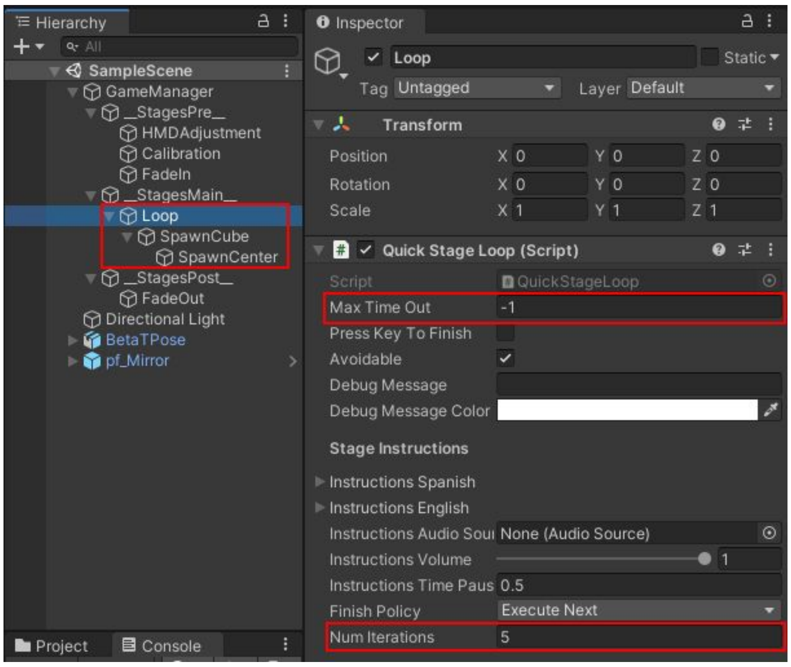
Position X 0 Y 1.5 Z 1
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Now modify the script, so the cube spawned appears in a random position inside a sphere with center at *_spawnCenter* and radius *_spawnRadius.*



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

using QuickVR;

Script de Unity | 0 referencias
public class StageSpawnCube : QuickStageBase
{

    public Transform _spawnCenter = null;
    public float _spawnRadius = 1.0f;

    22 referencias
    protected override IEnumerator CoUpdate()
    {
        yield return new WaitForSeconds(3);

        GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
        cube.transform.parent = transform;
        cube.transform.localScale = Vector3.one * 0.25f;

        cube.transform.position = _spawnCenter.position + (Random.insideUnitSphere * _spawnRadius);
    }

}
```

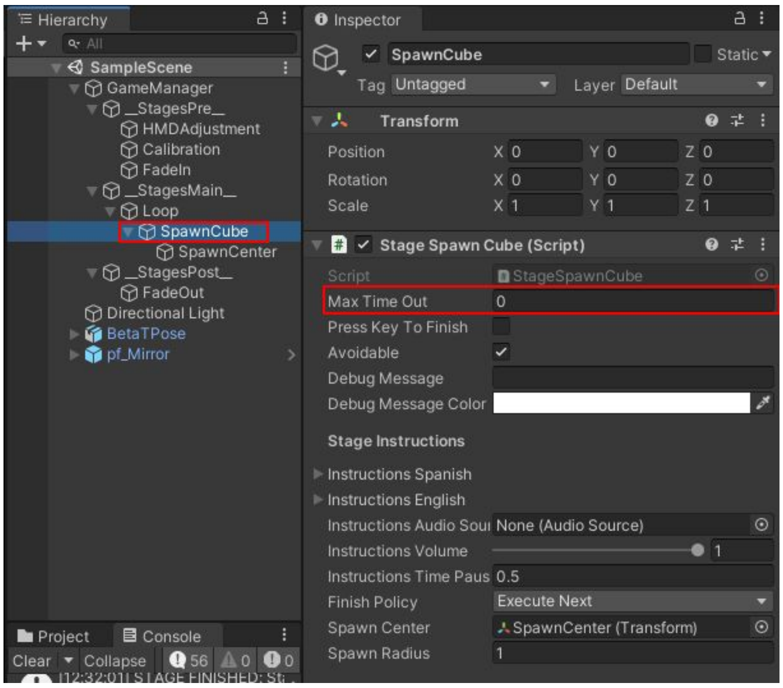
Now on the scene hierarchy, create a new child object of *__StagesMain__*. Name it *Loop* and add the component *QuickStageLoop* on it. Set the *Num Iterations* field to 5 and the *Max Time Out* field to -1. Finally, move the object *SpawnCube* to be child of *Loop.*



```
Hierarchy
+▾  All
SampleScene
  GameManager
    __StagesPre__
      HMDAdjustment
      Calibration
      FadeIn
    __StagesMain__
      Loop
        SpawnCube
          SpawnCenter
    __StagesPost__
      FadeOut
  Directional Light
  BetaTPose
  pf_Mirror
Project   Console
```

```
Inspector
Loop                               Static
Tag Untagged           Layer Default
Transform
Position      X 0    Y 0    Z 0
Rotation      X 0    Y 0    Z 0
Scale         X 1    Y 1    Z 1
Quick Stage Loop (Script)
Script                       QuickStageLoop
Max Time Out                 -1
Press Key To Finish          ☐
Avoidable                    ✓
Debug Message
Debug Message Color
Stage Instructions
Instructions Spanish
Instructions English
Instructions Audio Soui      None (Audio Source)
Instructions Volume          1
Instructions Time Paus       0.5
Finish Policy                Execute Next
Num Iterations               5
```

Before pressing play, go to the *SpawnCube* stage and set the *Max Time Out* field back to 0, or it will never end and the loop will get stuck on this stage.



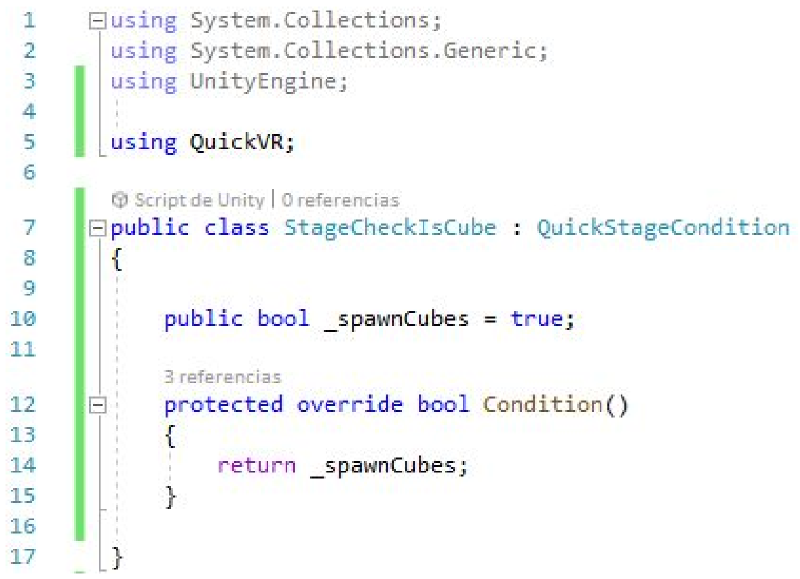
```
Hierarchy
+▾  All
SampleScene
  GameManager
    __StagesPre__
      HMDAdjustment
      Calibration
      FadeIn
    __StagesMain__
      Loop
        SpawnCube
          SpawnCenter
    __StagesPost__
      FadeOut
  Directional Light
  BetaTPose
  pf_Mirror
Project   Console
Clear ▾ Collapse   56  0  0
[12:32:01] STAGE FINISHED: St
```

```
Inspector
SpawnCube                          Static
Tag Untagged           Layer Default
Transform
Position      X 0    Y 0    Z 0
Rotation      X 0    Y 0    Z 0
Scale         X 1    Y 1    Z 1
Stage Spawn Cube (Script)
Script                       StageSpawnCube
Max Time Out                 0
Press Key To Finish          ☐
Avoidable                    ✓
Debug Message
Debug Message Color
Stage Instructions
Instructions Spanish
Instructions English
Instructions Audio Soui      None (Audio Source)
Instructions Volume          1
Instructions Time Paus       0.5
Finish Policy                Execute Next
Spawn Center                 SpawnCenter (Transform)
Spawn Radius                 1
```

Now press play and you will see that the logic of *SpawnCube* is repeated 5 times.

Creating an *if else* block using QuickStages

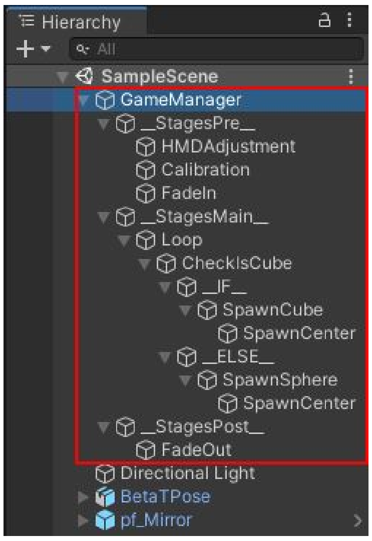
QuickStages can be also used to create an *if else* block. We will extend the example by adding a stage that will control if we spawn cubes or spheres depending on an attribute of that class. Create a new *C# Script* and name it *StageCheckIsCube.* Open this new script and make it a subclass of *QuickStageCondition.* In this case, it is mandatory to offer an implementation of the abstract member *Condition.* So do it as follows:



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

using QuickVR;

Script de Unity | 0 referencias
public class StageCheckIsCube : QuickStageCondition
{

    public bool _spawnCubes = true;

    3 referencias
    protected override bool Condition()
    {
        return _spawnCubes;
    }

}
```

On the scene hierarchy, create a new child of *Loop* and Name it *CheckIsCube.* Add the previously created component *StageCheckIsCube.* You will see that two child are automatically created, named __*IF*__ and __*ELSE*__. As you may guess, on the __*IF*__ branch we will set all the stages to be executed if the condition defined at *StageCheckIsCube* is true. On the contrary, the stages on the __*ELSE*__ branch will be executed.

So move the object *SpawnCube* into the __*IF*__ branch. Now we will create a new stage to have something to do in the __*ELSE*__ case. Following the same process that has been described before, create a new *QuickStage* that spawns a sphere instead of a cube. Add this component in a new child of the __*ELSE*__ object and name it *SpawnSphere.* The final hierarchy structure should look like this:



Finally press play to test it. Now we have spawned spheres instead of cubes. This is because the setting *IsCube* is set to false. Change the value of *Spawn Cubes* attribute of the *CheckIsCube* stage on the inspector to test the other branch. If you have followed the steps correctly, now it should spawn spheres instead of cubes.