Lecture 3
Wrangling Unity

98-127: Game Creation for People Who Want to Make Games (S19)

Written by Adrian Biagioli

Instructors:
Adrian Biagioli (abiagiol@andrew.cmu.edu)
Carter Williams (ncwillia@andrew.cmu.edu)

1 Objectives

By the end of this lesson you will be able to:

- Create C# scripts in the Unity editor, edit them, and apply them to GameObjects as Components
- Understand the basics of Unity's `MonoBehaviour`: `Start()`, `Update()`, and more
- Add easy-to-use interfaces to your component in the Unity Inspector, enabling artists and level designers to tweak parameters in your Unity code.
- Allow your scripts to inter-operate with other Components (of your own creation or Unity's)
- Use Prefabs to dynamically create GameObjects in your scripts

These lecture notes were written for **Unity 2018.3.3f1**.

2 Downloading the Base Code

You can download all of the base code for this lecture via the following Unitypackage. In these lecture notes, I've included the path to each script before every code sample (for example, the first code sample below is labeled `BasicFunctions` ▸ `StartTester.cs`, which means that you can find `StartTester.cs` in the `BasicFunctions` folder).

http://stage.gamecreation.org/StuCo/packages/lec03resources.unitypackage

See the lecture 2 notes for more info on how to import Unitypackages.

3 Cooking Components

In the last lecture, we went over how to use components created by your team members or Unity themselves to compose complex relationships between GameObjects. But what if we want to define our own custom behavior? Unity allows you to create your own components via a **C# Script**. C# is a programming language maintained by Microsoft. It is very similar to Java and should feel familiar to anyone who knows a C-based language. We will not cover how to code in this class; if you are interested in learning more about using C#, you can find plenty of resources online. We will try to write about important differences between C# and C/C++/Java/Python (languages that are used more often in classes at CMU), so do not fret if you have no experience with C# in particular. Read more about C# here.

To create a new component type, you need to create a C# Script in the project view. Open up the project view in Unity, right click on it, and select Create ⟩ C# Script. Alternatively you can navigate to Assets ⟩ Create ⟩ C# Script. Name your first script `StartTester.cs`. Let's make this script print the familiar "Hello, World" to the debugging console. Type the following in your editor:

`BasicFunctions‣StartTester.cs`

```
1  using UnityEngine;
2
3  public class StartTester : MonoBehaviour
4  {
5      // Start() is called exactly once when you launch the game
6      private void Start()
7      {
8          // Use Debug.Log(...) to log to the Console view
9          Debug.Log("Hello, World!");
10     }
11 }
```

This should look fairly familiar to those who have experience with Java or C#. On line 1 we see the `using UnityEngine` directive; this instructs C# to bring the `UnityEngine` classes in scope, so that we can use them[1]. Our class, called `StartTester`, represents the component that we want to create. You are **required** to name the class the same as the file. For example, because this component class is named `StartTester` we have to save it in `StartTester.cs`. The class derives from the `MonoBehaviour` base class[2]. **MonoBehaviour is the base class for all components**, and all components must derive from `MonoBehaviour` in this way[3].

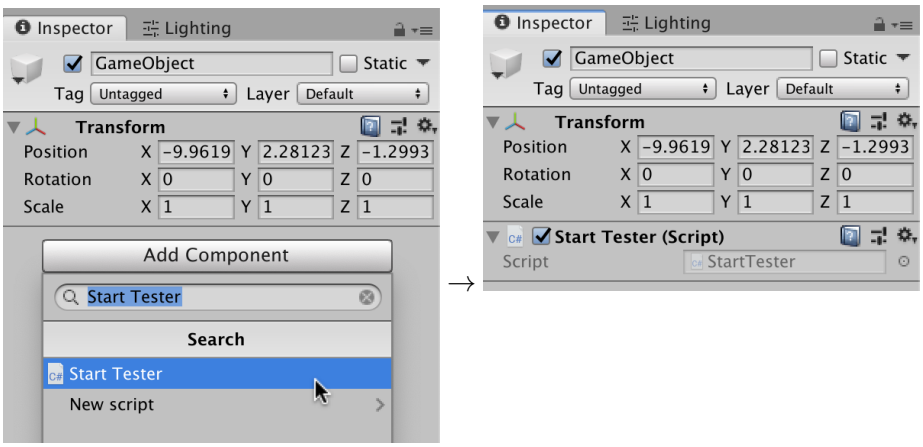
We have only added one function to this class, `void Start()`. The name `Start` is special for MonoBehaviours: `Start()` is called exactly once when this component has finished initialization. Inside of `Start()`, we call `Debug.Log(string)`, which is a Unity-provided equivalent of `Console.Log` or `printf`.

[1] You can think of the `using` keyword as similar to `import` in Java or `#include` in C/C++

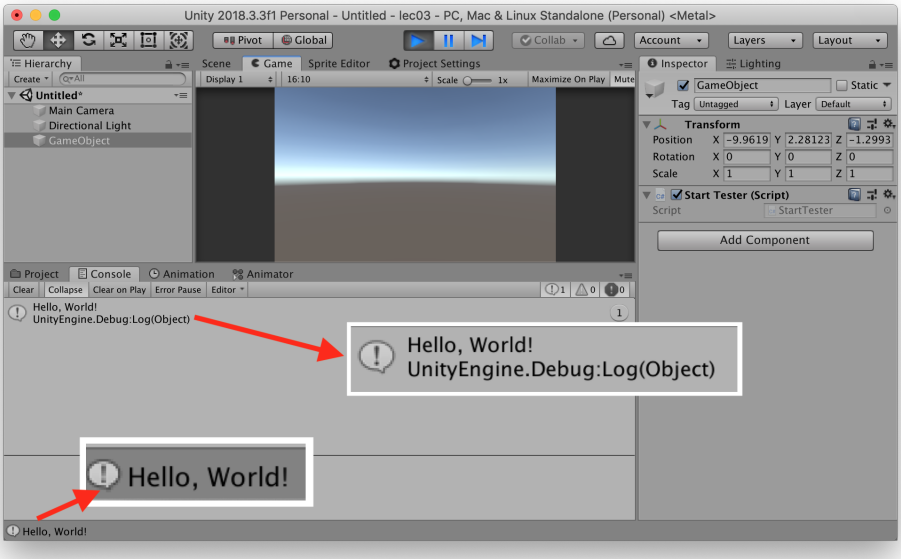
[2] The colon (`:`) operator works similarly to C++: it indicates in this case that `StartTester` *is a* `MonoBehaviour`. "`:`" can be compared to `extends` in Java. More info on this can be found in the C# documentation.

[3] The name `MonoBehaviour` comes from Mono, a cross-platform community implementation of the C# standard library and compiler (known together as Microsoft .NET). Unity uses a modified version of Mono for it's compiler.

Now let's add our newly-created component to a GameObject. Create an empty GameObject (GameObject > Create Empty) and select it in the Hierarchy view. Click on the Add Component button in the inspector, and search for "Start Tester." Alternatively you can drag and drop the `StartTester.cs` file into the inspector. Notice how Unity picked up the `CamelCasing` in the class name StartTester and expanded it to the more readable "Start Tester" name.



Hit the play button and you will see "Hello, World!" appear in the console previewer on the bottom left of the window. You can click on that message or navigate to Window > General > Console to open the Unity Console Window. The Console keeps track of all of the `Debug.Log` messages that you print in your scripts as well as generally useful debugging messages from Unity components.

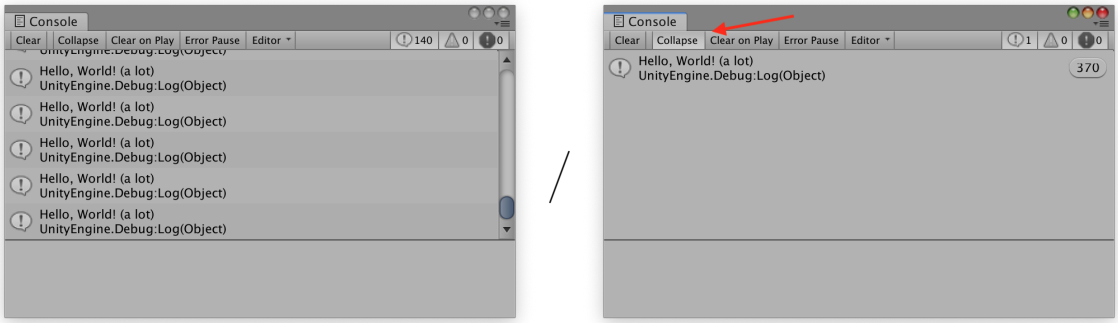


Our next example, the `UpdateTester`, replaces `Start` with a new function, called `Update()`. `Update` is called once per frame, unlike `Start` which is only called once. Most of your game logic goes inside of `Update`, and it is the most straightforward way to interact with the scene over time.

BasicFunctions▸UpdateTester.cs

```
using UnityEngine;

public class UpdateTester : MonoBehaviour
{
    // Update() is called once per frame
    void Update()
    {
        Debug.Log("Hello, World! (a lot)");
    }
}
```

Attach this script to the empty GameObject, and watch the console get spammed by "Hello, World!"s. This occurs because `Debug.Log` is being called once per frame, and there are around 60 frames per second. Therefore we get hundreds of messages in the Console window. You can click the Collapse button at the top of the Console to coalesce debug messages that come from the same line of code. This is incredibly helpful to parse the log when multiple `Debug.Log`s are being called per frame!



4 Adding Fields to Components

You may recall that Unity's built-in components are all *configurable* via the inspector. For example, we can modify a Box Collider's extents or a Character Controller's walk speed. You can make your own components configurable as well. There are two ways to do this:

1. Add a `public` variable to your component class
2. Add a `private` variable to your component class with `[SerializeField]` before the declaration. Special keywords in square brackets like SerializeField are called **attributes** in C#. Attributes allow you to give hints about how your code should be treated by the C# compiler and Unity's runtime; you can read more on them here. Read more on `[SerializeField]` in the Unity Scripting Reference.

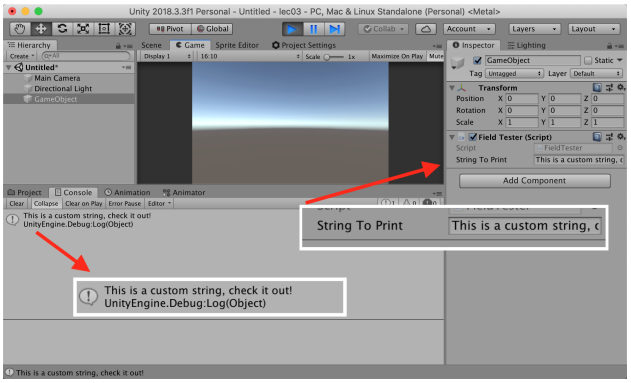
It is a matter of style whether or not you want to use `public` or `[SerializedField] private` variables; in general you should use `public` *only* if you intend for other scripts to modify that value. Unity will

automatically detect the type of each field and add a corresponding editor in the Unity inspector. Below is a simple modification on the `StartTester` that allows for a custom String to print:

`BasicFunctions‣FieldTester.cs`

```
using UnityEngine;

public class FieldTester : MonoBehaviour
{
    // Mark fields with [SerializeField] to allow
    // them to be edited in the inspector
    [SerializeField]
    private string StringToPrint;

    private void Start()
    {
        Debug.Log(StringToPrint);
    }
}
```

Here is what this script looks like in the Unity inspector: Notice how Unity once again picked up on the camel casing of `StringToPrint` and also detected the type of the variable (`string`). The input field that Unity displays is specific to this type. In fact, many different types are supported by Unity to be displayed via `[SerializedField]`. These types are known as **Serializable Types** [4] Here's a summary of which types are supported (Source: Unity Scripting Reference):



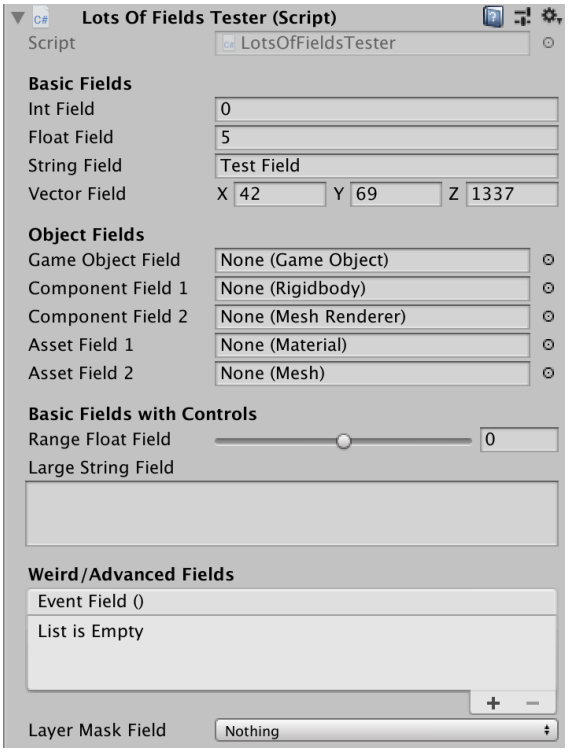
1. All classes inheriting from `UnityEngine.Object`, for example `GameObject`, `Component`, `MonoBehaviour`, `Texture2D`, and `AnimationClip`.
2. All basic data types like `int`, `string`, `float`, and `bool`.
3. Unity types: `Vector2`, `Vector3`, `Vector4`, `Quaternion`, `Matrix4x4`, `Color`, `Rect`, `LayerMask`...
4. Arrays of a serializable type (For example: `int[]`, `GameObject[]`)
5. Lists of a serializable type (For example: `List<Vector3>`)
6. `enum` types (more info here)
7. `structs`, as well as `classes` marked with the `[System.Serializable]` annotation.

The next example demonstrates different types of interesting Serializable types that are built in to Unity:

[4] The term *Serializable* is rooted in a C# concept known as *Serialization*. From the Microsoft C# documentation: "Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file." (Source) Unity is internally Serializing your game data to an asset file.

BasicFunctions‣LotsOfFieldsTester.cs

```
using UnityEngine;
using UnityEngine.Events;

public class LotsOfFieldsTester : MonoBehaviour
{
    [Header("Basic Fields")]
    [SerializeField]
    private int _IntField;
    [SerializeField]
    private float _FloatField = 5.0f;
    [SerializeField]
    private string _StringField = "Test Field";
    [SerializeField]
    private Vector3 _VectorField = new Vector3(42, 69, 1337);

    [Header("Object Fields")]
    [SerializeField]
    private GameObject _GameObjectField;
    [SerializeField]
    private Rigidbody _ComponentField1;
    [SerializeField]
    private MeshRenderer _ComponentField2;
    [SerializeField]
    private Material _AssetField1;
    [SerializeField]
    private Mesh _AssetField2;

    [Header("Basic Fields with Controls")]
    [SerializeField]
    [Range(-1.0f, 1.0f)]
    private float _RangeFloatField;
    [SerializeField]
    [TextArea]
    private string _LargeStringField;
    [Header("Weird/Advanced Fields")]
    [SerializeField]
    private UnityEvent _EventField;
    [SerializeField]
    private LayerMask _LayerMaskField;
}
```



Importantly, we can create fields for the basic types (`float`, `Vector3`, etc) but also for complex Unity types (components like `MeshRenderer`, GameObjects, Materials. . .). Also here I use the `[Header(string)]` attribute to add a nice bold header before certain controls. This is for aesthetic purposes only and does not affect gameplay, but can be nice to organize your fields for level designers. To set a default value, simply set the variable in your script equal to something (see the `_VectorField` for example)

5 Procedural Animation

One of the most common tasks that you need to do when creating components is to move GameObjects. The `transform` variable, available to all `MonoBehaviour`s, allows you to change the position, rotation, scale, and parent of any GameObject. For example, `transform.position = Vector3.zero;` translates the current script's GameObject to the origin (`Vector3`s are used to represent any kind of X/Y/Z position). The globally-accessible constant `Time.time` gives you the current time, in seconds, since the game launched. You can use this to build simple animations:

`BasicFunctions‣ClosedFormAnimation.cs`

```
using UnityEngine;

public class ClosedFormAnimation : MonoBehaviour
{
    [SerializeField]
    private float Radius = 1.0f;
    [SerializeField]
    // Vector3.zero == new Vector3(0,0,0)
    private Vector3 Center = Vector3.zero;
    [SerializeField]
    private float Speed = 1.0f;

    private void Update()
    {
        float s = Mathf.Sin(Time.time * Speed);
        float c = Mathf.Cos(Time.time * Speed);

        transform.position = Center + new Vector3(c * Radius, s * Radius, 0);
    }
}
```

The above script moves the GameObject it is attached to in a circle. The radius and center of the circle are given by variables in the inspector, as well as the speed of movement in radians per second. We use the Unity-provided Mathf library to get `Sin` and `Cos` functions, which are used to calculate the final position using basic trigonometry. Importantly, we use `Time.time` to find the position along the circle.

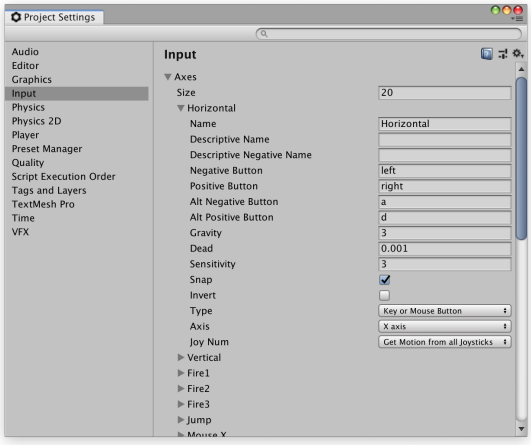
In real games, however, your components need to *respond* to "stimuli" that are outside of your control. For example, you may need to respond to user input which is totally unpredictable. Because of this, using `Time.time` to calculate an absolute position won't cut it. Instead, we would like to control the *velocity* of the parent GameObject over time. The below script is an example of this concept: instead of *setting* the position explicitly, we are *adding* to the "current" position.

BasicFunctions‣DeltaTimeAnimation.cs

```
1  using UnityEngine;
2
3  public class DeltaTimeAnimation : MonoBehaviour
4  {
5     public float Speed = 1.0f; // speed in m/s
6     public Vector3 Direction = new Vector3(1,0,0);
7     public bool localPosition = false;
8
9     // Update is called once per frame
10    private void Update()
11    {
12       Vector3 delta = Speed * Direction.normalized * Time.deltaTime;
13
14       if (localPosition)
15          transform.localPosition += delta;
16       else
17          transform.position += delta;
18    }
19 }
```

`Time.deltaTime` gives the time in seconds *since the last call to Update()*. It is used here on line 12 to ensure that the movement speed is independent of framerate. Let's say, for example, that you set the speed variable to 1 m/s. If you didn't multiply by `Time.deltaTime`, then assuming 60fps you would be moving at around 60 m/s! Even more worryingly, the GameObject would move faster on faster machines. Another way of thinking of this is via dimensional analysis; `Speed` is in $\frac{\text{m}}{\text{s}}$ and we want to know how many meters to move. By multiplying by `Time.deltaTime`, which is measured in seconds, we have $\frac{\text{m}}{\text{s}} \times \text{s} = \text{m}$, which is what we want. Also highlighted here is the difference between `transform.position` and `transform.localPosition`. `transform.position` changes the *absolute* position of the GameObject, ignoring any parenting. `transform.localPosition` represents the position of the GameObject *relative to* the parent GameObject (if the transform has no parent, the two values are the same).

5.1 Responding to User Input



One common task when building a game is to respond to some input by the player. This could be the Mouse, Keyboard, or a Gamepad. Unity requires that all *possible* inputs are first outlined in the **Input Manager**. Open the input manager via Edit ⟩ Project Settings... and select the Input submenu. The input manager consists of a set of named **Input Axes** that list out all of the possible inputs to your game. The default set of inputs, shown to the right, cover most common inputs that you would need in a simple game. For example, the "Horizontal" axis sets the so-called *positive button* to → / D and the *negative but-*

ton to ← / A. If you scroll down the input manager, you will find a second entry for "Horizontal" that is configured to respond to the X axis of a joystick. What this means is that the "Horizontal" axis will have a value of -1 if you press ← / A or if you hold the left stick on a gamepad to the left, and it will have a value of $+1$ if you press → / D or hold the left stick on a gamepad to the right. You can read more about each option in the Input Manager in the Unity Manual here.

The next script shows how to access these input axis values from within `Update()`. The operative command is `Input.GetAxis(string)`, which returns a `float` – usually ranging from -1 to $+1$ – of the value at that axis. The parameter matches the name of each axis that we configured in the Input Manager: in this case "Horizontal" and "Vertical." I like to allow the axis names to be configurable in Unity, so that if we need to change the Input Manager then we don't need to dive in to each script to fix the axis names.

`BasicFunctions▸SimpleMovement.cs`

```
1  using UnityEngine;
2
3  public class SimpleMovement : MonoBehaviour
4  {
5     [SerializeField]
6     private string _HorizontalMovementAxis = "Horizontal";
7     [SerializeField]
8     private string _VerticalMovementAxis = "Vertical";
9     [SerializeField]
10    private float _MovementSpeed = 1.0f;
11
12    private void Update()
13    {
14       float hoz = Input.GetAxis(_HorizontalMovementAxis);
15       float vrt = Input.GetAxis(_VerticalMovementAxis);
16
17       Vector3 mov = new Vector3(hoz, vrt, 0);
18
19       if (mov.sqrMagnitude > 1.0f)
20          mov.Normalize(); // make vector have length 1
21
22       transform.position += mov * _MovementSpeed * Time.deltaTime;
23    }
24 }
```

Once we have the input values (stored, again, in the range $-1 \leftrightarrow +1$ in the `hoz` / `vrt` variables) we convert them into a `Vector3` on line 17. On line 19, we protect against the case where the player holds both → + ↑ for example. In this case, `hoz` $=$ `vrt` $=$ 1, so the length of `mov` is (by pythagorean theorem) $\sqrt{1^2 + 1^2} = \sqrt{2}$, which is greater than 1. This would mean that the player can move faster by moving diagonally! This bug is actually pretty common in games with 2D movement (especially bad console ports). The solution is to call `mov.Normalize()`, which replaces `mov` with a vector in the same direction, but with length 1. If you attach `SimpleMovement.cs` to a GameObject, you can now control it with the arrow keys (or any other of the keys mentioned above.)

5.2 Accessing Other Components

What do you do if you want to access another component of the current GameObject? For example, our script might want to change the movement speed of a player character, the color of an object, etc. You can use the `GetComponent<Type>()` function (available to all `MonoBehaviours`) to access the component attached to the same GameObject as the caller with type `Type`. The following example illustrates how to use `GetComponent` to change the color of a GameObject. To do this we need to modify the GameObject's Material, which is stored in the *Mesh Renderer* component (as seen in lecture 2). Unity's components are named after the class names in C# scripts, just like your own components. Therefore we can deduce that the C# type of the "Mesh Renderer" component is `MeshRenderer`. Similarly "Box Collider" has the type `BoxCollider`, "Rigidbody 2D" has type `Rigidbody2D`, etc.

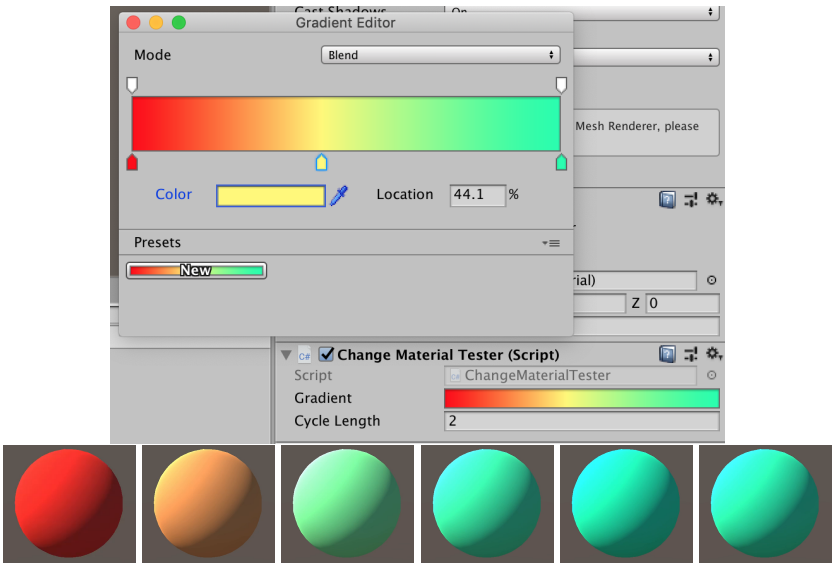
`BasicFunctions▸ChangeMaterialTester.cs`

```
using UnityEngine;

[RequireComponent(typeof(MeshRenderer))]
public class ChangeMaterialTester : MonoBehaviour
{
    [SerializeField]
    private Gradient _Gradient;
    [SerializeField]
    [Tooltip("Length in seconds to cycle through the gradient")]
    private float _CycleLength;

    // Note: This field isn't serialized, so we can't edit it in the inspector!
    private MeshRenderer _Renderer;

    private void Start()
    {
        // GetComponent<T> gets the component
        // with type T attached to the current GameObject.
        // If none exist, returns null
        _Renderer = GetComponent<MeshRenderer>();
    }

    private void Update()
    {
        float a = (Mathf.Cos(Time.time / _CycleLength * (2.0f * Mathf.PI)) +
            1.0f) / 2.0f;
        Color c = _Gradient.Evaluate(a);

        // _Renderer.material has type 'Material'
        // Material.color has type 'Color'
        _Renderer.material.color = c;
    }
}
```

This example also makes use of the built-in `Gradient` type to cycle between many colors over time. The value `float a` in `Update` cycles from 0 to 1 as time moves forward, and the gradient is *evaluated* at that value. The `Color` type is used to represent RGBA color values; we get at our `MeshRenderer`'s material via `MeshRenderer.material` and further are able to change the color via `Material.color`. How do I know all of this? Am I a god, and have just memorized all of the keywords? Of course not: I simply went to Unity's **Scripting Reference**, which can be accessed at

https://docs.unity3d.com/ScriptReference

For example, you can find a reference for the `MeshRenderer` here and a reference for `Gradient` here. The Scripting API reference pages are an *absolutely essential reference* for Unity programmers–if you want to learn all of Unity's ins and outs, expect to look at this often! Anyway, here's what this script looks like in the inspector. The Gradient type happens to have a really awesome interactive editor:



Top: The gradient popup menu featured in `ChangeMaterialTester`.
Bottom: Result when attached to a sphere.

6 Talking to the Physics Engine

At one point when making almost any game, you'll have to deal with collisions. You need to think about collisions when a projectile leaves a cannon, a pong ball bounces off the paddle, Mario hits the bottom of an item box, Sonic collects a ring, etc. In many cases, you want to *react* to collisions (for example: when the player collides with a coin GameObject you might want to destroy the coin, spawn a shiny particle effect, and increment a coin counter). In the last lecture, we talked about **Colliders**. Colliders are special components that talk with Unity's physics engine and mark a GameObject as "collidable." You also use Colliders to define the *shape* of the collidable object (which can be different than what is rendered on screen). There are a few different types of colliders: the *Box Collider* and *Sphere Collider* are shaped like a Box and Sphere, and the *Mesh Collider* allows you to specify a mesh to collide with. As we discussed in the last lecture, you

can additionally add a *Rigidbody* component to have Unity simulate the GameObject, applying forces such as Gravity and reacting to collisions realistically.

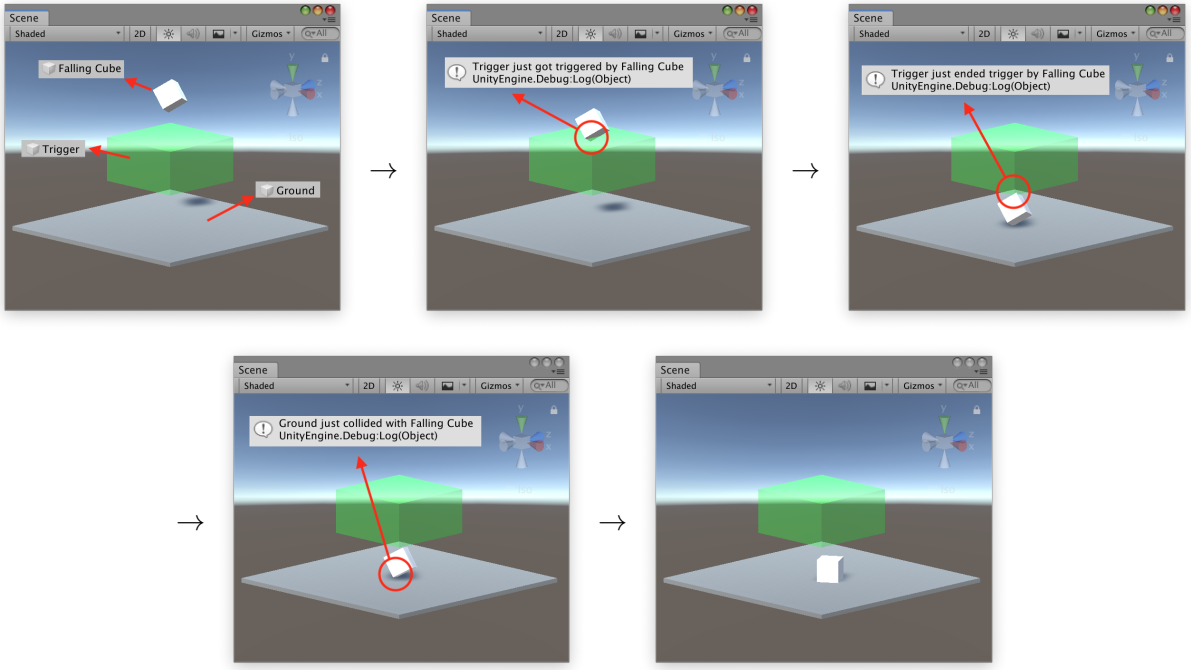
If you want to respond to collisions with your own logic, create a C# script as normal. Instead of using `Start` or `Update` to check for collisions, we can use the built-in `OnCollisionEnter` / `OnCollisionExit` / `OnTriggerEnter` / `OnTriggerExit` functions. Recall from the last lecture that *triggers* are colliders with the "Is Trigger" option enabled in the inspector. Triggers do not affect GameObjects physically (for example they don't block rigidbodies from overlapping with them). Instead, triggers are meant to be used by your scripts for logic. For example, in a Mario game you would want to set the coins as triggers so that they do not affect Mario's velocity when you collect them. However you would want to set the item boxes as Colliders because Mario should be able to bump his head against them.

`PhysicsExamples▸CollisionDetectionTester.cs`

```
1  using UnityEngine;
2
3  public class CollisionDetectionTester : MonoBehaviour
4  {
5      private void OnTriggerEnter(Collider other)
6      {
7          Debug.Log(gameObject.name + " just got triggered by " +
               other.gameObject.name);
8      }
9
10     private void OnTriggerExit(Collider other)
11     {
12         Debug.Log(gameObject.name + " just ended trigger by " +
               other.gameObject.name);
13     }
14
15     private void OnCollisionEnter(Collision collision)
16     {
17         Debug.Log(gameObject.name + " just collided with " +
               collision.gameObject.name);
18     }
19
20     private void OnCollisionExit(Collision collision)
21     {
22         Debug.Log(gameObject.name + " just ended collision with " +
               collision.gameObject.name);
23     }
24 }
```

To demonstrate this script, below is a scene with three cubes. The bottom cube, called "Ground," has a Box Collider with *Is Trigger* disabled. The middle (transparent green) cube, named "Trigger," has a Box Collider with *Is Trigger* enabled. The top cube, named "Falling Cube," has a Box Collider and a Rigidbody attached. The ground and trigger cubes both have the above `CollisionDetectionTester` component. When we run the script, `OnTriger{Enter,Exit}` is called on the trigger when the falling cube enters and

exits it, and `OnCollisisonEnter` is called on the ground when the falling cube collides with it. Because the falling cube never stops colliding with the ground, `OnCollisionExit` is not called.



This is also the first time that I've used the `gameObject` variable. `gameObject` is available to all `MonoBehaviour`s and provides a pointer to the "owner" of this instance of the `MonoBehaviour`. The type of `gameObject` is `UnityEngine.GameObject`.

If you are making a 2D game, there is a completely different set of components that you need to use for Physics. For example, in a 2D game, instead of using a *Box Collider* component you should use a *Box Collider 2D*. There is a 2D version of every physics component: *Rigidbody* and *Rigidbody 2D*, *Sphere Collider* and *Circle Collider 2D*, etc. Similarly, when responding to 2D collision we need to use 2D variants of the collision response functions: `OnTriggerEnter2D`, `OnTriggerExit2D`, `OnCollisionEnter2D`, `OnCollisionExit2D`. Unity is built this way because under the hood, the 3D physics system uses NVIDIA PhysX, which is optimized for 3D games. The 2D physics system uses the Box2D Physics Engine, which is purpose-built for 2D games. On the next page is a 2D variant of the collision detection tester that will respond to 2D physics components. **Do not use the 3D versions of OnCollision/OnTrigger functions with 2D colliders!**

PhysicsExamples▸CollisionDetectionTester2D.cs

```
using UnityEngine;

public class CollisionDetectionTester2D : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D other)
    {
        Debug.Log("Just got triggered by GameObject called " +
            other.gameObject.name);
    }

    private void OnTriggerExit2D(Collider2D other)
    {
        Debug.Log("Just ended trigger by GameObject called " +
            other.gameObject.name);
    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        Debug.Log("Just collided with GameObject called " +
            collision.gameObject.name);
    }

    private void OnCollisionExit2D(Collision2D collision)
    {
        Debug.Log("Just ended collision with GameObject called " +
            collision.gameObject.name);
    }
}
```

7 Using UnityEvents

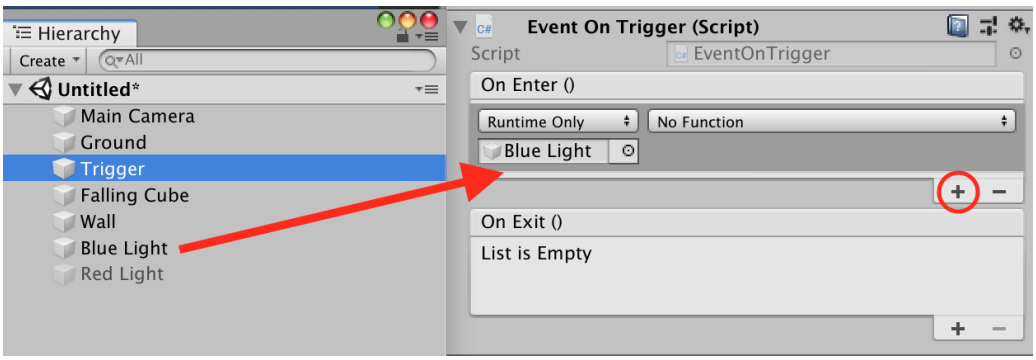
Triggers are a great opportunity to talk about a very useful feature of Unity's scripting called **UnityEvents**. UnityEvents are similar to `event`s in C# and function pointers in C/C++. They are an implementation of the Command Design Pattern, which is sort of like a *callback*. UnityEvents allow you to *store an action into a variable, such that you can execute that action at any time*. Even better, these actions are editable in the inspector, just like any other field!

The next script is an example of how to use UnityEvents to build Game Logic directly in the inspector. We have two editable `UnityEvent`s that are meant to respond to trigger enter and exit events. In the `OnTrigger` events, we then call `UnityEvent.Invoke()`, which actually executes the event. Note that if no actions are assigned to either UnityEvent, they are set to `null`. So we need to check for `null` before calling `Invoke()`.

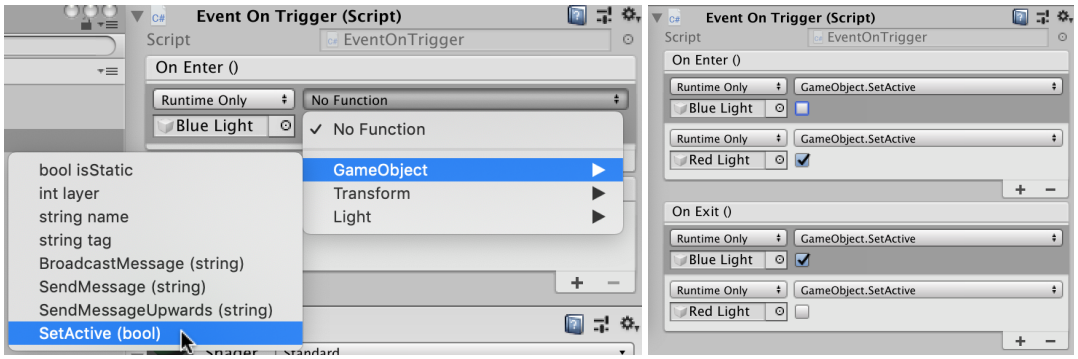
PhysicsExamples‣EventOnTrigger.cs

```
using UnityEngine;
using UnityEngine.Events; // Required for UnityEvent type

public class EventOnTrigger : MonoBehaviour
{
    [SerializeField]
    private UnityEvent _OnEnter;
    [SerializeField]
    private UnityEvent _OnExit;

    private void OnTriggerEnter(Collider other)
    {
        if(_OnEnter != null)
            _OnEnter.Invoke();
    }

    private void OnTriggerExit(Collider other)
    {
        if(_OnExit != null)
            _OnExit.Invoke();
    }
}
```

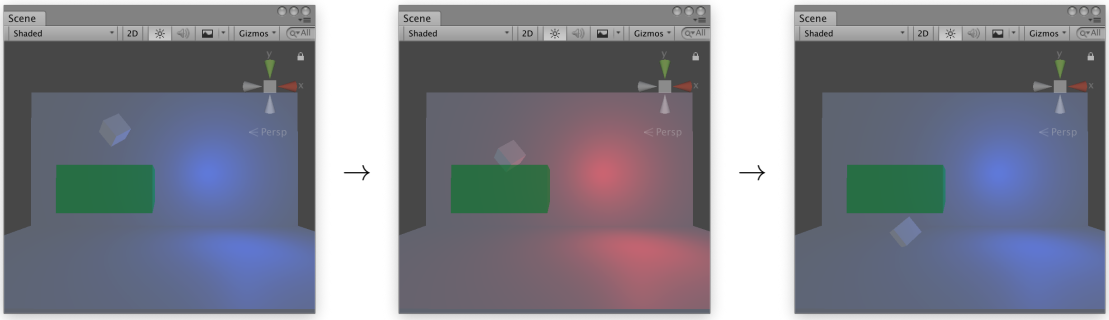
To demonstrate this I've set up two lights, a red and blue light. I'd like to enable the red light when a trigger is being triggered, and blue when it is not. I can use the `EventOnTrigger` script to do this. UnityEvents have an interactive editor in the inspector that allows you to select which function you want to call on which GameObject. Here I call the function `GameObject.SetActive()` to enable and disable the red and blue lights in response to each event. Normally this would have required me to write a new script, but UnityEvents let me do this right in the inspector! UnityEvents are a powerful tool, and allow you to build simple behaviors without touching a line of code.



Click on the ✚ button to add a new action to the UnityEvent. All actions are executed simultaneously upon calling `UnityEvent.Invoke()` in your script.



Here we configure the event to enable and disable the lights. You need to specify a *target* GameObject to call something on. Here we are calling `GameObject.SetActive`, but we can call functions in the target's `GameObject`, `Transform`, or any component.

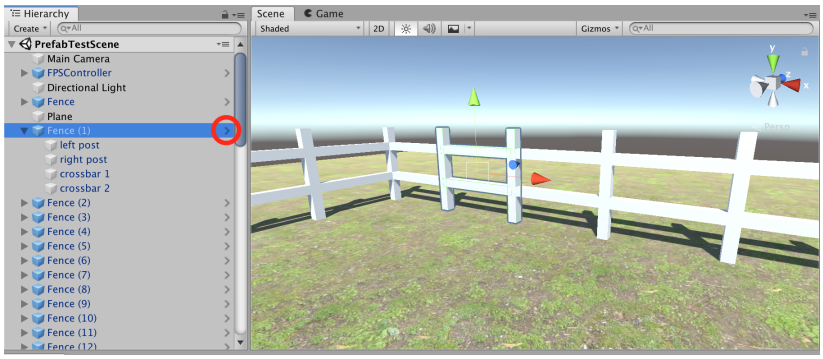


When a cube activates the trigger (the green cube), the lights are enabled and disabled as expected.

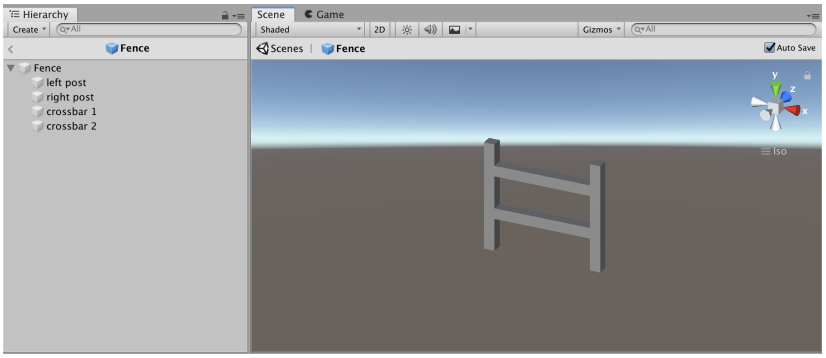
8 Prefabs

Game worlds tend to have many objects that are repeated very often. For example, you might need to spawn in dozens of the exact same enemy in a level. Obviously you wouldn't want to rebuild the enemy GameObject every time you wanted to spawn it in. One way to solve this issue is to build the enemy once and duplicate it (⌘ or Ctrl + D). This works at first, but what if you want to change a detail of your enemy after duplicating them? You would have to re-do your change for each instance.

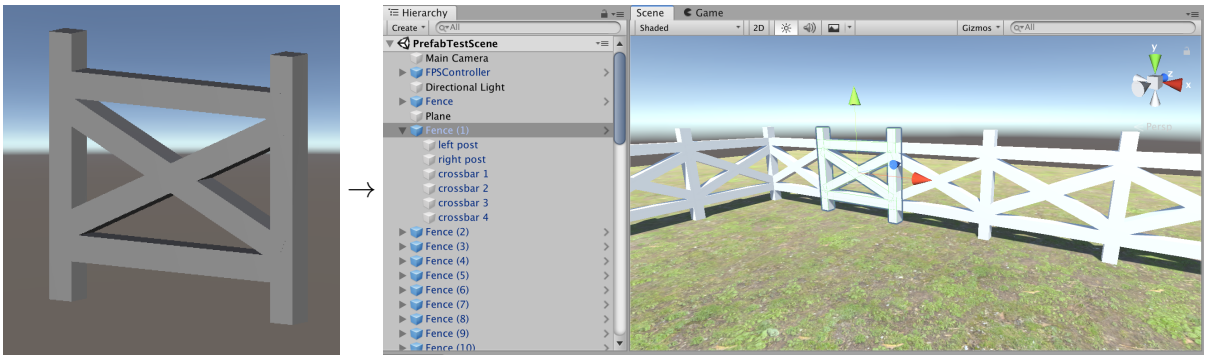
Unity addresses this common problem via **Prefabs**. Prefabs are *GameObjects that are saved to a file*. You can create a prefab by dragging a GameObject from the hierarchy view onto the project view. To see prefabs in action, open the scene located in the base code at `ScriptingIntro‣PrefabExamples‣PrefabTestScene.unity`. Here I've created a small environment with a First Person Controller that you can walk around in (if you want to review how to configure the First Person Controller, see lecture 2). The environment is surrounded by a fence. However, instead of making the fence out of one mesh, I've made it out of many copies of a GameObject for each individual link in the fence.



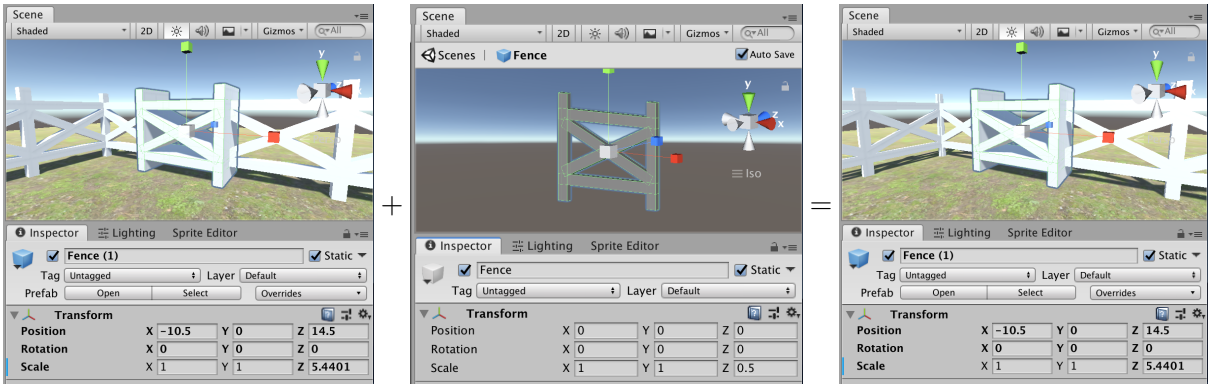
In the above picture we've selected one individual fence link. Notice how, in the hierarchy view, the fence GameObjects have a blue box next to them. This is because each of the fence GameObjects is **linked** to a prefab. The prefab itself is located at `ScriptingIntro‣PrefabExamples‣Fence.asset`. We can modify the prefab by double clicking on the prefab file OR clicking on the small arrow next to a fence GameObject at the right side of the hierarchy view (circled above). Unity will then "jump into" the prefab so that you can edit it directly:



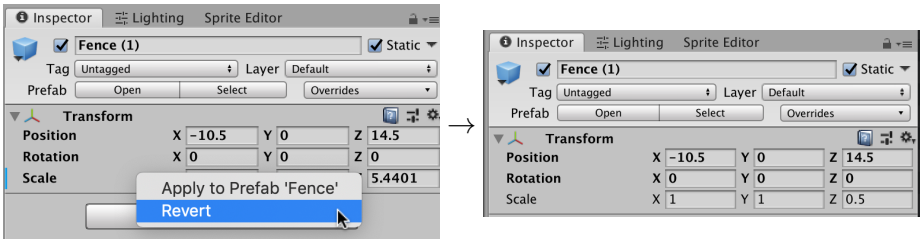
Let's pretend that while working on this game, your team lead complained that the fences look too boring. They suggested adding more details to the fences. If we edit the prefab by adding a crossbar, then the changes will apply to each instance throughout the scene:



Keep in mind that if we change any of the instances of the fence link in our scene, those changes will **override** the prefab version. For example, if we change the Z-scale of one of our fence link instances, then change the Z-scale in the prefab to something else, the instance will *retain the changes* you made to it:



Notice how in the above images, the scale field of the instance (see the left and right picture above) has a small blue mark to the left of it. This means that the scale of this object overrides the prefab. You can also see which fields are overridden because they are displayed in bold. You can reset the Z-scale to what is stored in the prefab by right clicking on the field and selecting "Revert." Alternatively you can apply your change to the main prefab by selecting "Apply to Prefab . . . "



8.1 Instantiating Prefabs at Runtime

You can also use prefabs to *spawn* GameObjects into your scene. You could use this to spawn waves of enemies into your levels over time, for example. There is a command called `Instantiate<Type>`, available to all `MonoBehaviours`, that creates a copy of whichever object you pass into it. The below script spawns a prefab (specified via the field `_PrefabToSpawn`, which is of type `GameObject`) into the scene whenever the player presses a button. In this case the default is to use `Fire1` as the axis, which is the left mouse button in the default Input Manager configuration (see "Responding to User Input" above). In the following example we use `Input.GetButtonDown(string)`, which is `true` only on the first frame that the user presses that button.

PrefabExamples▸SpawnPrefabOnButton.cs

```
using UnityEngine;

public class SpawnPrefabOnButton : MonoBehaviour
{
    [SerializeField]
    private string _ButtonToPress = "Fire1";
    [SerializeField]
    private GameObject _PrefabToSpawn;
    [SerializeField]
    private Transform _TransformToSpawnAt;

    private void Update()
    {
        if(Input.GetButtonDown(_ButtonToPress))
        {
            var go = Instantiate<GameObject>(_PrefabToSpawn);
            go.transform.position = _TransformToSpawnAt == null ?
                transform.position : _TransformToSpawnAt.position;
            go.transform.rotation = _TransformToSpawnAt == null ?
                transform.rotation : _TransformToSpawnAt.rotation;
        }
    }
}
```

You might also want to spawn a prefab every N seconds. A great way to do this is by using *Coroutines*. Coroutines make clever use of C#'s asynchronous programming features to *defer execution* of a function to a later frame. See this Unity manual page for more information on coroutines:

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

The next example uses Coroutines to automatically instantiate a prefab every `_SpawnIntervalSeconds` seconds. This works because on line 26 we defer execution of `SpawnPrefabCoroutine()` to a later frame.

PrefabExamples▸SpawnPrefabInterval.cs

```
using System.Collections;
using UnityEngine;

public class SpawnPrefabInterval : MonoBehaviour
{
    [SerializeField]
    private float _SpawnIntervalSeconds = 1.0f;
    [SerializeField]
    private GameObject _PrefabToSpawn;
    [SerializeField]
    private Transform _TransformToSpawnAt;
```

```
12
13      private void Start()
14      {
15          StartCoroutine(SpawnPrefabCoroutine());
16      }
17
18      private IEnumerator SpawnPrefabCoroutine()
19      {
20          while(true)
21          {
22              var go = Instantiate<GameObject>(_PrefabToSpawn);
23              go.transform.position = _TransformToSpawnAt == null ?
                    transform.position : _TransformToSpawnAt.position;
24              go.transform.rotation = _TransformToSpawnAt == null ?
                    transform.rotation : _TransformToSpawnAt.rotation;
25
26              yield return new WaitForSeconds(_SpawnIntervalSeconds);
27          }
28      }
29  }
```