UnityNaiveGuide

Alternatively checkout *.pdf version.

Preface

Probably the main question that you might have once reaching this page and seing lots of text which varies in quality: “Is it one more Unity Guide?”
- Yes. There are so many damn good resources on the web that sometimes it is difficult to find them all until you search for something very specific. I will put here all of those as well as some other tests I do in spare time (if I am even capable to compete with those bright minds who help improving my everydays code). I hope that set of practises here can be a good starting poing for someone. The text will have many alterations as long as I or other contributors learn new things about Unity.

I’d like to mention that not all the tests and recommendations can fit your project. **I belive that every line is written here makes you more suspicios, resulting in validating the information you are after on here** and then contributing to this guide with your clarifications. Highly appreciate this approach, so lets learn together!

It is important to mention that the tests that are provided in this guide should be considered quite sceptically. I believe that if you really want to compare two or more operations on which performance is better, then it is fair to compare on the level of instructions. However, I don’t have the access to Unity’s source code as well as I am not a specialist to go that deep to disassemble code. That is why all I can do is running some tests, which, sometimes, can be far from what you might face in production.

PS, if you want to tell me at some point of reading that some of my examples are far-fetched and/or stupid, that they don’t pass peer review in big teams – I am very proud for your team and still sure that not all the teams are like yours!

Table of Contents

Script Templates

It is a very handy to adjust your script templates to make files that already have all the required setup. Simple use case can be legal notice that you can also do with code snippets. . . , but isn't having a script template easier? When the size of your project turns really big, it becomes quite handy. In addition to legal notice you can customize your template to make preset for DOTS or any other custom systems, e.g. NightCache C# Script Template.

Add Custom templates

There are a few ways to make your script templates. The most convinient, easiest is to create a folder called `ScriptTemplates` under your `Assets`. Inside you should create, for instance, `81-C# Script-NewBehaviourScript.cs.txt`. This template should override your default 'C# Script' in the Right-Click menu in 'Project' Unity Editor window. '81' is the number that is used for ordering in that menu, same as order that you put in `CreateAssetMenu()` when make ScriptableObject. If you put 82, 83 or any other, with the same ending, your script will appear above/below as a *duplicate*, if the number is the same, but you, lets say, put "`81-B# Script`", then it should appear above. This way you don't override the script, that is why you should have the part "`81-C# Script`" identical. However, you are free to change `NewBehaviourScript`, that is how the default name of the script will look like.

Note: these script templates do not override script creation if you create with "Add Component" -> "New Script" via Inspector Window. Unfortunately, if you need this, it is available in the Unity app directory, see below paths corresponding to different platforms.

The number of scripts that you/your studio uses can be different. If it is extensive, you can group your options with "___", **double** underscore. For example, you can have your scripts under **Fancy Studios Templates** like:

```
81-Fancy Studios Templates__C# Class-NewClass.cs.txt
81-Fancy Studios Templates__C# Interface-NewInterface.cs.txt
81-Fancy Studios Templates__C# Scriptable Object-NewScriptableObjectScript.cs.txt
```

A few words about how your template *can* look. I normally avoid `target` and use only `serializedObject`, but it can be a good illustration. Also, for Editor script it is fine if file name and class names are different. So when you create your Editor script you can name it the same as `MonoBehaviour` class, they are in different assemblies.

```
//////////////////////////////
///  Not production-ready  ///
//////////////////////////////

using UnityEditor;
```

```
namespace NAMESPACE
{
    [CustomEditor(typeof(#SCRIPTNAME#))]
    public class #SCRIPTNAME#Editor : Editor
    {
        private void OnEnable()
        {
            #NOTRIM#
        }

        /// <summary>
        /// <see cref="#SCRIPTNAME#"/>
        /// </summary>
        public override void OnInspectorGUI()
        {
            var #SCRIPTNAME_LOWER# = (#SCRIPTNAME#)target;

            EditorGUI.BeginDisabledGroup(true);
            EditorGUILayout.PropertyField(serializedObject.FindProperty("m_Script"));
            EditorGUI.EndDisabledGroup();

            // var serializedPropertyMyInt = serializedObject.FindProperty("");
            //serializedObject.ApplyModifiedProperties();
        }
    }
}
```

- **#SCRIPTNAME#** – Replaces it with the filename on script creation.
- **#SCRIPTNAME_LOWER#** – Similar as the one above, but changes the first uppercase letter to lowercase. Before if you name your file with lowercase as first letter, it was removing all the lowercase letters until it reaches uppercase. Now Unity adds 'my' prefix. So if filename is `scriptMB` then it swaps `#SCRIPTNAME_LOWER#` to `myScriptMB`.
- **#NOTRIM#** – Lets parser know not to clean this empty line, e.g. empty function. However, it might not have any effect. It can preserve many tabulations which, I recon, you don't need.

Original Templates

Locations of original files: * MacOS * Unity: * UnityHub: `*/UnityInstalls/*/Unity.app/Contents/Resource`
* Windows * Unity: * UnityHub: * Linux * Unity: * UnityHub:

Handle Manually

Side By Side Comparison

`Shader.PropertyToID("")` and `Animator.StringToHash("")`

The offcial pages for Animator.StringToHash and Shader.PropertyToID, where it clearly say "IDs are used for optimized setters and getters..." and "using property identifiers is more efficient than passing strings..." respectively. So you can consider avoiding strings as the parameter when you call any of these.

Probably a Better Approach

IDEs, e.g. as Rider, automatically recognise similar cases in your project and suggest to hash `strings` into `ints`. However, over time you might encounter duplicates of the same parameters and it is probably better to have a special place for all of them. The example below also allows you to control all the names in one file. There is also an option to make a configue file or ScriptableObject file for that, but the simplest example is below:

```
namespace Test.Properties
{
    using UnityEngine;

    public static class AnimatorProperties
    {
        public static readonly int RotateYAnimationClip = Animator.StringToHash("RotateY");
        public static readonly int RotateZAnimationClip = Animator.StringToHash("RotateZ");
    }
    public static class ShaderProperties
    {
        public static readonly int ColorShaderProperty = Shader.PropertyToID("_Color");
        public static readonly int TintShaderProperty = Shader.PropertyToID("_Tint");
    }

    // OR
    [CreateAssetMenu(fileName = "Properties", menuName = "ScriptableObjects/Properties")]
    public class PropertiesSO : ScriptableObject
    {
        public readonly int RotateYAnimationClip = Animator.StringToHash("RotateY");
        public readonly int RotateZAnimationClip = Animator.StringToHash("RotateZ");

        public readonly int ColorShaderProperty = Shader.PropertyToID("_Color");
        public readonly int TintShaderProperty = Shader.PropertyToID("_Tint");
    }
}
```

Example for Animators:

```
namespace Tests.HashingStrings.Animators
{
    using UnityEngine;

    using HashedProperties;

    public class AnimationChangerMB : MonoBehaviour
    {
        [SerializeField]
        protected GameObject sphere1;
        [SerializeField]
        protected GameObject sphere2;

        protected bool Swapper = false;
        protected Animator Sphere1Animator;
        protected Animator Sphere2Animator;

        protected void Awake()
        {
            Sphere1Animator = sphere1.GetComponent<Animator>();
            Sphere2Animator = sphere2.GetComponent<Animator>();
        }

        protected void Update()
        {
            if (!Input.GetKeyDown(KeyCode.Tab))
                return;

            if (Swapper)
            {
                Sphere1Animator.Play(AnimatorProperties.RotateZAnimationClip);
                Sphere2Animator.Play(AnimatorProperties.RotateYAnimationClip);
            }
            else
            {
                Sphere1Animator.Play(AnimatorProperties.RotateYAnimationClip);
                Sphere2Animator.Play(AnimatorProperties.RotateZAnimationClip);
            }
            Swapper = !Swapper;
        }
    }
}
```

Performance

- Test #1 – Execute with string e.g. `Sphere1Animator.Play("RotateZ");`
- Test #2 – Execute with locally cached int value;
- Test #3 – Execute with static int from static class;
- Test #4 – Same as above, but value was cached before loop;
- Test #5 – Execute with int from ScriptableObject class;
- Test #6 – Same as above, but value was cached before loop.

Editor

Times	Test #1	Test #2	Test #3	Test #4	Test #5	Test #6
1	0.00014849	0.00007170	0.00000700	0.00001120	0.00000738	0.00001053
1	0.00000515	0.00000505	0.00000321	0.00000467	0.00000475	0.00000306
10	0.00000666	0.00000404	0.00000669	0.00000592	0.00000386	0.00000384
100	0.00002740	0.00002500	0.00001750	0.00002276	0.00001755	0.00001788
1000	0.00033476	0.00015649	0.00020201	0.00015632	0.00015598	0.00023105
10000	0.00240798	0.00155762	0.00133051	0.00140078	0.00166718	0.00153883
100000	0.01805707	0.01114511	0.01113095	0.01122464	0.01120038	0.01113555
1000000	0.14642538	0.09188678	0.09118677	0.09149084	0.09296595	0.09170776
10000000	1.38090207	0.90337447	0.90877167	0.91157750	0.91057538	0.94460000

IL2CPP MacOS Intel 64-bit

Times	Test #1	Test #2	Test #3	Test #4	Test #5	Test #6
1	0.00000887	0.00000081	0.00000183	0.00000223	0.00000204	0.00000220
1	0.00000457	0.00000221	0.00000232	0.00000203	0.00000191	0.00000214
10	0.00001018	0.00000407	0.00000394	0.00000421	0.00000356	0.00000364
100	0.00002641	0.00001685	0.00001701	0.00001661	0.00001667	0.00001705
1000	0.00024906	0.00014647	0.00016511	0.00016372	0.00016302	0.00014549
10000	0.00235644	0.00149591	0.00156760	0.00150800	0.00151446	0.00147985
100000	0.01664717	0.01035973	0.00960241	0.00968604	0.00914258	0.01052656
1000000	0.10926833	0.06973652	0.07076260	0.07580751	0.07004646	0.07040648
10000000	1.00485862	0.63289100	0.63487495	0.63104334	0.63236006	0.63263087

Example for Shaders:

If you make different shaders with the same parameter name, e.g. `"_Color"`, there should be no problems to **re**use it for different shaders. Unity recalculates value, so there should be no collision. You can try it yourself: just make two spheres with different materials & shaders. Have a property of the same name and type in both shaders, the one you'll access with `.SetColor()`, `.SetFloat()` etc.

```
namespace Tests.HashingStrings.Shaders
{
```

```
    using UnityEngine;

    using HashedProperties;

    public class ColorChangerMB : MonoBehaviour
    {
        [SerializeField]
        protected GameObject sphere1;
        [SerializeField]
        protected GameObject sphere2;

        protected Material Sphere1Material;
        protected Material Sphere2Material;

        protected void Awake()
        {
            Sphere1Material = sphere1.GetComponent<Renderer>().sharedMaterial;
            Sphere2Material = sphere2.GetComponent<Renderer>().sharedMaterial;
        }

        protected void Update()
        {
            if (!Input.GetKeyDown(KeyCode.Space))
                return;
            var color1R = Random.Range(0f, 1f);
            var color1G = Random.Range(0f, 1f);
            var color1B = Random.Range(0f, 1f);

            Sphere1Material.SetColor(ShaderProperties.ColorShaderProperty,
                new Color(color1R, color1G, color1B, 1f));

            var color2R = Random.Range(0f, 1f);
            var color2G = Random.Range(0f, 1f);
            var color2B = Random.Range(0f, 1f);

            Sphere2Material.SetColor(ShaderProperties.ColorShaderProperty,
                new Color(color1R, color1G, color1B, 1f));
            // Sphere2Material.SetColor(ShaderProperties.ColorShaderProperty,
            //     new Color(color2R, color2G, color2B, 1f));
        }
    }
}
```

Just make two spheres, two materials and two shaders for them. Put the code below in your shaders for `Sphere_1` and `Sphere_2` respectively.

```
Shader "Unlit/Sphere_1" // Shader "Unlit/Sphere_2"
```

```
{
    Properties { _Color ("Main Color", Color) = (1,1,1,1) }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            struct appdata
            {
                float4 vertex : POSITION;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
            };

            float4 _Color;

            v2f vert (appdata v) {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                return _Color;      // Shader "Unlit/Sphere_1"
                return 1 - _Color;  // Shader "Unlit/Sphere_2"
            }
            ENDCG
        }
    }
}
```

Performance

- Test #1 – Execute with string e.g. `Sphere1Material.SetColor("_Color", SomeColour);`
- Test #2 – Execute with locally cached int value;

- Test #3 – Execute with static int from static class;
- Test #4 – Same as above, but value was cached before loop;
- Test #5 – Execute with int from ScriptableObject class;
- Test #6 – Same as above, but value was cached before loop.

Editor

Times	Test #1	Test #2	Test #3	Test #4	Test #5	Test #6
1	0.00000831	0.00007536	0.00002213	0.00000792	0.00000742	0.00000848
1 again	0.00000742	0.00000383	0.00000318	0.00000312	0.00000321	0.00000183
10	0.00000503	0.00000458	0.00000430	0.00000523	0.00000454	0.00000515
100	0.00002602	0.00001307	0.00001299	0.00001517	0.00001715	0.00001714
1000	0.00026692	0.00012060	0.00011721	0.00015282	0.00015758	0.00011767
10000	0.00206858	0.00093458	0.00114345	0.00094270	0.00093802	0.00093176
100000	0.01528322	0.00805394	0.00815373	0.00802989	0.00792480	0.00802744
1000000	0.12462365	0.06748622	0.06878386	0.07070566	0.06860883	0.06701352
10000000	1.18080147	0.65498759	0.65808497	0.65136666	0.65934728	0.65718837

IL2CPP MacOS Intel 64-bit

Times	Test #1	Test #2	Test #3	Test #4	Test #5	Test #6
1	0.00001183	0.00000180	0.00000127	0.00000151	0.00000284	0.00000312
1 again	0.00001055	0.00000188	0.00000180	0.00000172	0.00000167	0.00000217
10	0.00000690	0.00000307	0.00000323	0.00000564	0.00000318	0.00000355
100	0.00002648	0.00001332	0.00006365	0.00001357	0.00001278	0.00001380
1000	0.00023381	0.00011365	0.00011388	0.00019378	0.00013290	0.00011368
10000	0.00233917	0.00120554	0.00119545	0.00116674	0.00118861	0.00117835
100000	0.01656655	0.00874680	0.00864122	0.00908912	0.00848267	0.00857170
1000000	0.11665278	0.05940517	0.05452891	0.05607068	0.05616178	0.05674499
10000000	1.00733784	0.49475148	0.50794711	0.49516667	0.49861623	0.49419712

Opinion

it is hard to imagine that you have a real reason to update your shader property in the same frame for more than 10-100 materials. If you do so, probably you have to consider how to reuse your materials. If you can reuse your materials Unity could properly use batching. It is clearly 1.5-2x boost if avoiding `string`s as a parameter.

Practically you can see that it is fine to have a separate class to keep your properties. From management perspective it is just easier to change things in one place. Static class or a SO is a preference thing, it only can be quite annoying to assign refereneces if you don't make a singleton or don't automate assigning for your SO. Making a local variable for this case is not useful much.

Test Setup: there were 100x100 Objects spawned, each had a sphere mesh, one MeshRenderer and 1-16 `BoxColliders`. Actually also Sphere collider, forgot to remove it, but doesnt matter, I dont update it.

Prefab structure - ROOT_GO - UnityUpdateComponentsEnableMB(Clone) - Sphere (**1** `MeshRenderer`, **1-16** `BoxCollider(s)`, `MeshFilter` and `SphereCollider`) - UnityUpdateComponentsEnableMB(Clone) - Sphere (**1** `MeshRenderer`, **1-16** `BoxCollider(s)`, `MeshFilter` and `SphereCollider`) - etc, 100x100 of UnityUpdateComponentsEnableMB.

Below are the code samples I used for testing:

```
// UnityUpdateGOSetActiveEachMB.cs
protected void Update()
{
    // T1: it is fair to make a variable same is for T2 and T3
    // sphere is SphereRenderer.gameObject
    var isEnabled = Time.frameCount % 2 == 0;
    sphere.SetActive(isEnabled);
}
// UnityUpdateComponentsEnableMB.cs
protected void Update()
{
    // T2 and T3 were commented out and were tested separately
    var isEnabled = Time.frameCount % 2 == 0;

    // T2: SphereRenderer is MeshRenderer
    SphereRenderer.enabled = isEnabled;

    // T3: SphereBoxColliders is List<BoxCollider>
    for (int i = 0; i < numberOfComponents; ++i)
    {
        SphereBoxColliders[i].enabled = isEnabled;
    }
}
```

In update I 1. Enable/disable whole game object, 2. Mesh renderer, 3. Forloop every box collider

Test: 1, number of `BoxColliders` is 1 |№| Test | FPS | |-| - | ———:| |1: **1** `BoxCollider` | | | | | `GameObject.SetActive()` | 12.8-14 | | | 1 `MeshRenderer` | 33-35 | | | 1 `BoxCollider` for-loop | 19.5-20.5 | |2: **2** `BoxColliders` | | | | | `GameObject.SetActive()` | 10.5-11 | | | 1 `MeshRenderer` | 35-37 | | | | 2 `BoxCollider` for-loop | 12.4-12.8 | |3: **4** `BoxColliders` | | | | | `GameObject.SetActive()` | 7.6-8.8 | | | 1 `MeshRenderer` | 32-35 | | | 4 `BoxCollider` for-loop | 7.9-8.6 | |4: **8** `BoxColliders` | | | | | `GameObject.SetActive()` | 5.2-5.6 | | | 1 `MeshRenderer` | 32-35 | | | 8 `BoxCollider` for-loop | 4.8-5 | |5: **16** `BoxColliders`| | | | |

`GameObject.SetActive()` | 3.4-3.5 | | | 1 `MeshRenderer` | 31-32 | | | 16 `BoxCollider` for-loop | 1.9-3.1 |

Results Interpretation: when there are just a few components attached to a GameObject, it is more costly to disable the whole GameObject. Disabling particular conponents would be better. However, when there are more Components, *roughly 6+* per GO, it seems that iterating them in for-loop seems to be less effective than disabling using GameObject.SetActive(false); because it is done better by Unity on the engine level.

The second test evaluates a scenario when Enabled/Disabled GameObject has children in the hierarchy versus Enabling/Disabling particular components. This time it is `MeshRenderer`. Below is the structure of prefab:

Prefab structure - ROOT_GO - UnityUpdateGOHasChildren(Clone) - Sphere (**1** `MeshRenderer`, `MeshFilter` and `SphereCollider`) - SphereInstantiated(Clone) (**1** `MeshRenderer`, `MeshFilter` and `SphereCollider`); - SphereInstantiated(Clone) (**1** `MeshRenderer`, `MeshFilter` and `SphereCollider`); - SphereInstantiated(Clone) (**1** `MeshRenderer`, `MeshFilter` and `SphereCollider`); - etc, Total 1-16 SphereInstantiated based on Test №, see below - UnityUpdateGOHasChildren(Clone) - Sphere (**1** `MeshRenderer`, `MeshFilter` and `SphereCollider`) - SphereInstantiated(Clone) (**1** `MeshRenderer`, `MeshFilter` and `SphereCollider`); - SphereInstantiated(Clone) (**1** `MeshRenderer`, `MeshFilter` and `SphereCollider`); - SphereInstantiated(Clone) (**1** `MeshRenderer`, `MeshFilter` and `SphereCollider`); - etc, Total 1-16 SphereInstantiated based on Test №, see below - etc, 100x100 of UnityUpdateGOHasChildren.

№	Test	FPS
1: **1** `BoxCollider`		
	Parent `GameObject.SetActive()`	8-10
	1 `MeshRenderer` for-loop	21.7-23.3
2: **2** `BoxCollider`s		
	Parent `GameObject.SetActive()`	5.9-7.7
	2 `MeshRenderer` for-loop	13.6-16.5
3: **4** `BoxCollider`s		
	Parent `GameObject.SetActive()`	4.2-4.3
	4 `MeshRenderer` for-loop	8.9-9.1
4: **8** `BoxCollider`s		
	Parent `GameObject.SetActive()`	2.6-2.7
	8 `MeshRenderer` for-loop	4.8-5.2
5: **16** `BoxCollider`s		
	Parent `GameObject.SetActive()`	~0.2-1*
	16 `MeshRenderer` for-loop	2.6-2.7

~0.2-1* the game just didn't manage at that point and I can conclude that having

abusing all the potential of Hierarchies is dangerous.

`Instantiate<T>()`

Well, everyone knows that it is a bad idea to abuse that function. Just make sure you don't do as well this. *Yes, I saw that many times, so don't make that expensive function take even more time for no reason, all of those can be other parameters of* `Instantiate<T>()`.

```
protected void Spawner()
{
    for (int i = 0; i < YOUR_LIST.Count; ++i)
    {
        var component = Instantiate<YOUR_TYPE>(prefab);
        component.transform.parent = transform;
        component.transform.position = Vector3.zero;
        // something else, mb rotation. Again this is how NOT to do
    }
}
```