Bryce Dixon's CommonUtils

CommonUtils is a collection of common utility scripts I use across my projects and other packages.

Installation and Setup

The package will be available on the Unity Asset Store.

When using classes from these scripts in your projects, I would highly recommend *not* including the `BryceDixon.CommonUtils` namesapce and instead only including the `BryceDixon` namespace (unless otherwise stated) as some classes may share names with commonly used classes and namespaces within the `UnityEngine` namespace.

Note: This package depends on Denis Rizov's NaughtyAttributes package which can be found on the Unity Asset Store.

Usage Overview

All included code is thoroughly commented and descriptions for each class and method can be found by hovering over them in most IDEs (only confirmed in Visual Studio 2019), so this section will function mostly as a quick-start guide.

For more detail on any section, class, or method, read the comments provided directly above it in the code.

AnimatorExtensions

This class resides in the `BryceDixon.CommonUtils.Extensions` namespace which should be included in your script for it to function properly.

AnimatorExtensions adds a few methods to make interfacing with parameters of the UnityEngine.Animator class easier. Primarily, they allow for more generic Animator interfacing code by only attempting to set a parameter of the Animator if one exists with that name.

- `Animator.TrySetParameter(string name, [float, bool, int] value)` works as a generic alternative to most of the extension methods in this class.
    - I'd highly recommend *just* using this one unless you want to be more explicit.
- `Animator.TrySet{Type}(string name, {Type} value)` is the main family of extension methods in the class including the following:
    - `Animator.TrySetFloat(string name, float value)`
    - `Animator.TrySetBool(string name, bool value)`
    - `Animator.TrySetInteger(string name, int value)`
- `Animator.TrySetTrigger(string name)` and `Animator.TryResetTrigger(string name)` will call `Animator.SetTrigger(name)` and `Animator.ResetTrigger(name)`

Gizmo

Gizmo is an abstract base class for creating simple components for drawing UnityEngine.Gizmos.

To create your own, inherit from this base class and implement the `DrawShape()` method.

Existing derived classes include the following:

- Gizmo_Cube
- Gizmo_Frustum
- Gizmo_Line
- Gizmo_Sphere
- Gizmo_TransformLine
- Gizmo_WireCube
- Gizmo_WireSphere

SafeGet

SafeComponent

This acts as a wrapper around `GameObject.GetComponent` which adds cache. Its use is intended to solve two failures of traditional use of `GameObject.GetComponent` where either:

1. `GameObject.GetComponent` is used throughout code frequently, which leads to poor performance; or
2. `GameObject.GetComponent` is only used during initialization with the result cached, which can lead to a stale state if the cached Component is destroyed or intended target is created after the call to `GameObject.GetComponent`

SafeComponents

Achieves the same goals as SafeComponent, but gets/caches *all* components of a given type on the owning object.

Note that SafeComponents will only attempt to find more components if its cache is empty, so a different solution must be used if you're intending to repeatedly create/destroy components.

SafeFindObject

Achieves the same goals as SafeComponent, but wraps around `Object.FindObjectsOfType` instead of `GameObject.GetComponent`.

SafeFindObjects

Achieves the same goals as SafeComponent, but wraps around `Object.FindObjectsOfTypes` instead of `GameObject.GetComponent`.

This suffers from the same shortcomings of SafeComponents, specifically regarding its laziness in finding new objects if it still has existing objects cached.

SchmittTrigger

Threshhold based float to boolean converter based on the Schmitt trigger electronic component.

SchmittTriggers can be useful for preventing rapid on-off cycling when conditions are near their change threshhold

For example, if we check an analogue trigger against an exact value like this: `if (Input.GetAxis("Horizontal") > 0.5f)`

That could cause a rapid switching between true and false if the player were to hold the stick near 0.5f. Instead, we can make a SchmittTrigger to provide some buffered range to the tested input

This can also be useful for testing object positions, distances, etc. For example, we could make an AI that becomes agressive when the player is within 5 units, but only becomes passive again once they're more than 10 units away.

A SchmittTrigger will automatically be converted to a bool when attempting to evaluate it as a conditional.

Settings

A CRTP base class inteded for creating `UnityEngine.ScriptableObject`s for developers to store settings about their project and systems.

For example, the CommonUtils Loading system uses a Settings class called LoadingSettings to store which scene should be used as a loading screen.

Another example could be creating a derived class for exposing a list of variables that can be modified by systems designers; eg: character jump height, movement speed, etc.

Here's a bare-bones example of how to create your own Settings class:

```
using BryceDixon;

[SettingsPath("Settings/Movement")]
public class MovementSettings : CommonUtil.Settings<MovementSettings>
{
  [SerializeField]
  private float m_jumpHeight;
  public float jumpHeight { get => m_jumpHeight; }
  [SerializeField]
  private float m_moveSpeed;
  public float moveSpeed { get => m_moveSpeed; }

#if UNITY_EDITOR
  [UnityEditor.MenuItem("CommonUtils/Settings/Create Movement Screen Settings",
priority = 1)]
  private static new void CreateSettings()
  {
    CommonUtil.Settings<MovementSettings>.CreateSettings();
  }
#endif
}
```

Elsewhere, we can retrieve that data using `MovementSettings.GetSettings()`. Eg: `MovementSettings.GetSettings().moveSpeed` to get the set movement speed

Singleton

A CRTP base class inteded for creating Singletons. Deriving from this is a way to effectively have static data while also benefitting from Unity's `MonoBehaviour` methods (eg: `Start`, `Update`, etc.)

You should *not* put your singleton scripts on objects; their instances will be created automatically for you the first time you use it. For example:

```
using BryceDixon;

public class SystemController : Singleton<SystemController>
{
  public int m_val;

  private void Update()
  {
    // Do something with m_val
  }
}
```

```
public class SomeObject : MonoBehaviour
{
  private void Update()
  {
    // This will create an instance of SystemController the first time it's called
    SystemController.Get().m_val = 5;
  }
}
```

Timer

Class used for in-place timing operations. Timers are identified by their stacktrace, so we need to provide an extra value in certain situations to get unique Timers.

```
  void Update()
  {
    // Timers are an easy way of creating repeatable events without any
initialization
    // This conditional will be true every 10 seconds
    if (Timer.Every(10.0f))
    {
      Debug.Log("10 seconds have passed!");
    }

    for (int i = 0; i < 4; ++i)
    {
      float time = 1.0f * i;
```

```
        if (Timer.Every(time, (ulong)i))
        {
          Debug.Log(time + " second(s) have passed!");
        }
      }
    }
```

Version

Simple semantic versioning struct. Useful for distinguishing builds.

```
(new Version(5)).ToString(); // "5.0.0"
(new Version(2, 1)).ToString(); // "2.1.0"
(new Version(1, 4, 21)).ToString(); // "1.4.21"
(new Version(Version.ReleaseType.Beta, 4, 1, 2)).ToString(); // "b 4.1.2"

new Version(1, 5) >= new Version(1, 4, 1); // true
new Version(1, 3) >= new Version(1, 4, 1); // false
new Version(2) >= new Version(1, 4, 1); // true
new Version(Version.ReleaseType.Beta, 1) >= new Version(Version.ReleaseType.Alpha,
1); // true
new Version(Version.ReleaseType.Beta, 1) >= new Version(Version.ReleaseType.Alpha,
2); // false
```

Loading

LoadingSettings

Some settings for the Loading system. To use the Loading system:

1. Create the settings with the menu option: CommonUtils -> Settings -> Create Loading Screen Settings
2. Add your "Loading" scene to the build
3. Make sure there is exactly one object with the `BryceDixon.CommonUtils.SceneLoader` component in your "Loading" scene.
4. Add another scene (for this example, I'll assume it's named "MyScene") to the build
5. Select the file `Assets/Resources/Settings/Loading`
6. Provide your "Loading" scene and a failsafe scene.
7. The failsafe scene will be used if you try to load a scene that doesn't exist. I would recommend the title screen or a dedicated error scene.
8. Use `BryceDixon.CommonUtils.SceneLoader.LoadScene("MyScene")` to load a scene through the "Loading" scene you set.

Scene

A set of useful Scene-related methods.

`SceneExists(string name)` will tell you if a scene with the given name exists in the build.

`GetCurrentScene()` will get you the active scene. I mostly made this because I found the built-in call unreasonably long.

SceneLoader

This component has some functionality for easily loading other scenes. For example, you could make a button that calls `LoadScene()` on an attached `SceneLoader` to have it load the specified scene by name.