FREE eBook

LEARNING unity3d

Free unaffiliated eBook created from
Stack Overflow contributors.

#unity3d

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: unity3d

It is an unofficial and free unity3d ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official unity3d.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with unity3d

Remarks

Unity provides a cross platform game development environment for developers. Developers can use C# language and/or JavaScript syntax based UnityScript for programming the game. Target deployment platforms can be switched easily in the editor. All core game code stays same except some platform dependent features. A list of all the versions and corresponding downloads and release notes can be found here: https://unity3d.com/get-unity/download/archive.

Versions

Version	Release Date
Unity 2017.1.0	2017-07-10
5.6.2	2017-06-21
5.6.1	2017-05-11
5.6.0	2017-03-31
5.5.3	2017-03-31
5.5.2	2017-02-24
5.5.1	2017-01-24
5.5	2016-11-30
5.4.3	2016-11-17
5.4.2	2016-10-21
5.4.1	2016-09-08
5.4.0	2016-07-28
5.3.6	2016-07-20
5.3.5	2016-05-20
5.3.4	2016-03-15
5.3.3	2016-02-23
5.3.2	2016-01-28

Version	Release Date
5.3.1	2015-12-18
5.3.0	2015-12-08
5.2.5	2016-06-01
5.2.4	2015-12-16
5.2.3	2015-11-19
5.2.2	2015-10-21
5.2.1	2015-09-22
5.2.0	2015-09-08
5.1.5	2015-06-07
5.1.4	2015-10-06
5.1.3	2015-08-24
5.1.2	2015-07-16
5.1.1	2015-06-18
5.1.0	2015-06-09
5.0.4	2015-07-06
5.0.3	2015-06-09
5.0.2	2015-05-13
5.0.1	2015-04-01
5.0.0	2015-03-03
4.7.2	2016-05-31
4.7.1	2016-02-25
4.7.0	2015-12-17
4.6.9	2015-10-15
4.6.8	2015-08-26
4.6.7	2015-07-01

Version	Release Date
4.6.6	2015-06-08
4.6.5	2015-04-30
4.6.4	2015-03-26
4.6.3	2015-02-19
4.6.2	2015-01-29
4.6.1	2014-12-09
4.6.0	2014-11-25
4.5.5	2014-10-13
4.5.4	2014-09-11
4.5.3	2014-08-12
4.5.2	2014-07-10
4.5.1	2014-06-12
4.5.0	2014-05-27
4.3.4	2014-01-29
4.3.3	2014-01-13
4.3.2	2013-12-18
4.3.1	2013-11-28
4.3.0	2013-11-12
4.2.2	2013-10-10
4.2.1	2013-09-05
4.2.0	2013-07-22
4.1.5	2013-06-08
4.1.4	2013-06-06
4.1.3	2013-05-23
4.1.2	2013-03-26

Version	Release Date
4.1.0	2013-03-13
4.0.1	2013-01-12
4.0.0	2012-11-13
3.5.7	2012-12-14
3.5.6	2012-09-27
3.5.5	2012-08-08
3.5.4	2012-07-20
3.5.3	2012-06-30
3.5.2	2012-05-15
3.5.1	2012-04-12
3.5.0	2012-02-14
3.4.2	2011-10-26
3.4.1	2011-09-20
3.4.0	2011-07-26

Examples

Installation or Setup

Overview

Unity runs on Windows and Mac. There is also a Linux alpha version available.

There are 4 different payment plans for Unity:

1. **Personal** - Free *(see below)*
2. **Plus** - $35 USD per month per seat *(see below)*
3. **Pro** - $125 USD per month per seat - After subscribing to the Pro plan for 24 consecutive months, you have the option to stop subscribing and keep the version you have.
4. **Enterprise** - Contact Unity for more information

*According to EULA: Companies or incorporated entities that had a turnover in excess of US$100,000 in their last fiscal year must use **Unity Plus** (or a higher license); in excess of*

*US$200,000 they must use **Unity Pro** (or Enterprise).*

Installing

1. Download the Unity download assistant.
2. Run the assistant and choose the modules you want to download and install, such as Unity editor, MonoDevelop IDE, documentation, and desired platform build modules.

If you have an older version, you can update to the latest stable version.

If you want to install Unity without Unity download assistant, you can get the **component installers** from Unity 5.5.1 release notes.

Installing Multiple Versions of Unity

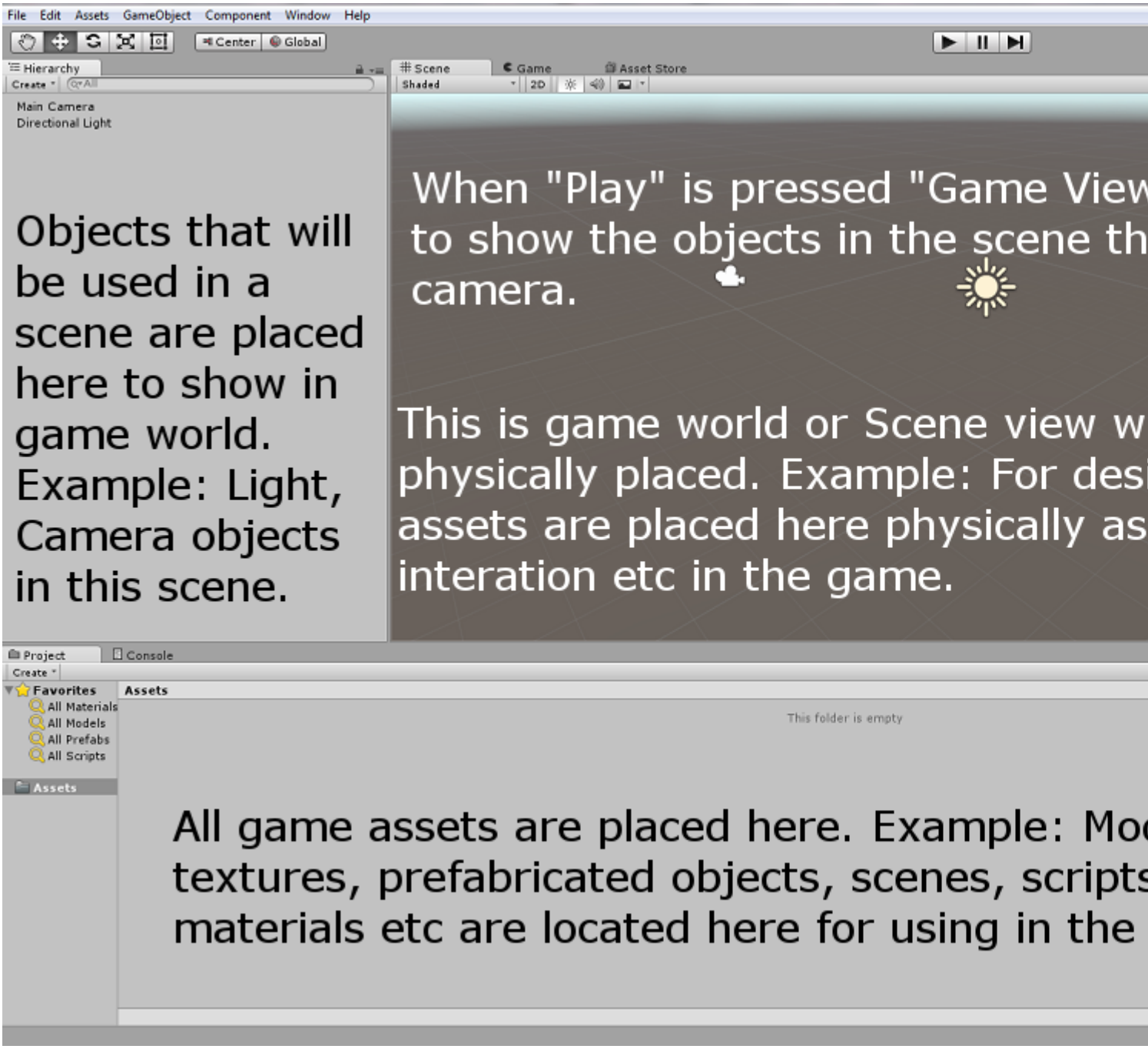
It is often necessary to install multiple versions of Unity at the same time. To do so:

- On Windows, change the default install directory to an empty folder that you have previously created such as `Unity 5.3.1f1`.
- On Mac, the installer will always install to `/Applications/Unity`. Rename this folder for your existing install (e.g. to `/Applications/Unity5.3.1f1`) before running the installer for the different version.
- You can hold `Alt` when launching Unity to force it to let you choose a project to open. Otherwise the last project loaded will attempt to load (if available) and it may prompt you to update a project you do not want updated.

Basic editor and code

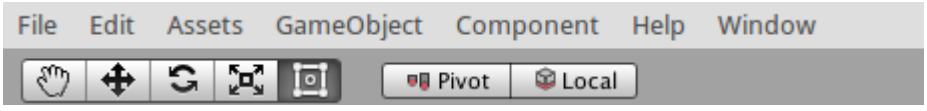
Layout

Unity basic editor will look like below. Basic functionalities of some default windows/tabs are described in the image.



Linux Layout

There is a little difference in menu layout of linux version, like the screenshot below,



Basic Usage

Create an empty `GameObject` by right clicking in the Hierarchy window and select `Create Empty`. Create a new script by right clicking in the Project window and select `Create` > `C# Script`. Rename it as needed.

When the empty `GameObject` is selected in the Hierarchy window, drag and drop the newly created

script in the Inspector window. Now the script is attached to the object in the Hierarchy window. Open the script with the default MonoDevelop IDE or your preference.

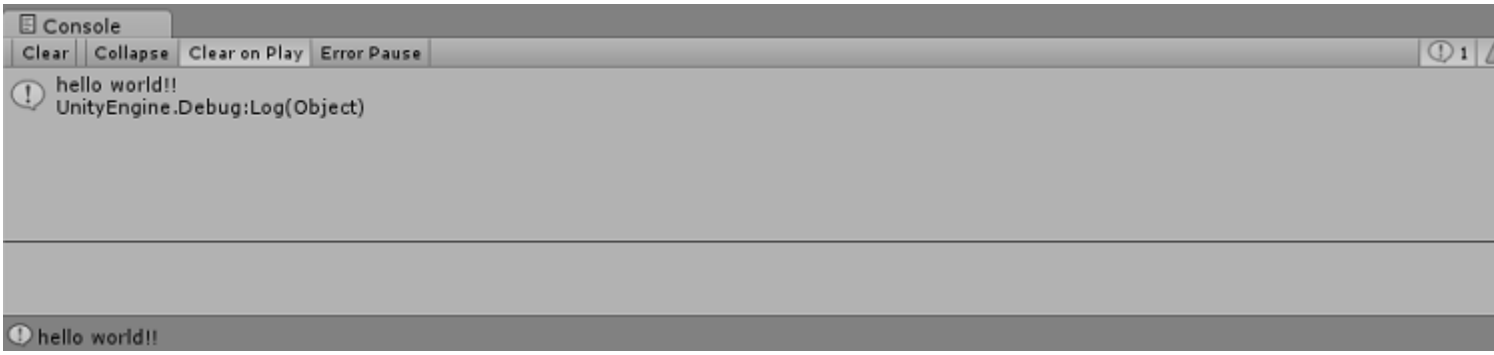
Basic Scripting

Basic code will look like below except the line `Debug.Log("hello world!!");`.

```
using UnityEngine;
using System.Collections;

public class BasicCode : MonoBehaviour {

    // Use this for initialization
    void Start () {
        Debug.Log("hello world!!");
    }

    // Update is called once per frame
    void Update () {

    }
}
```

Add the line `Debug.Log("hello world!!");` in the `void Start()` method. Save the script and go back to editor. Run it by pressing **Play** at the top of the editor.

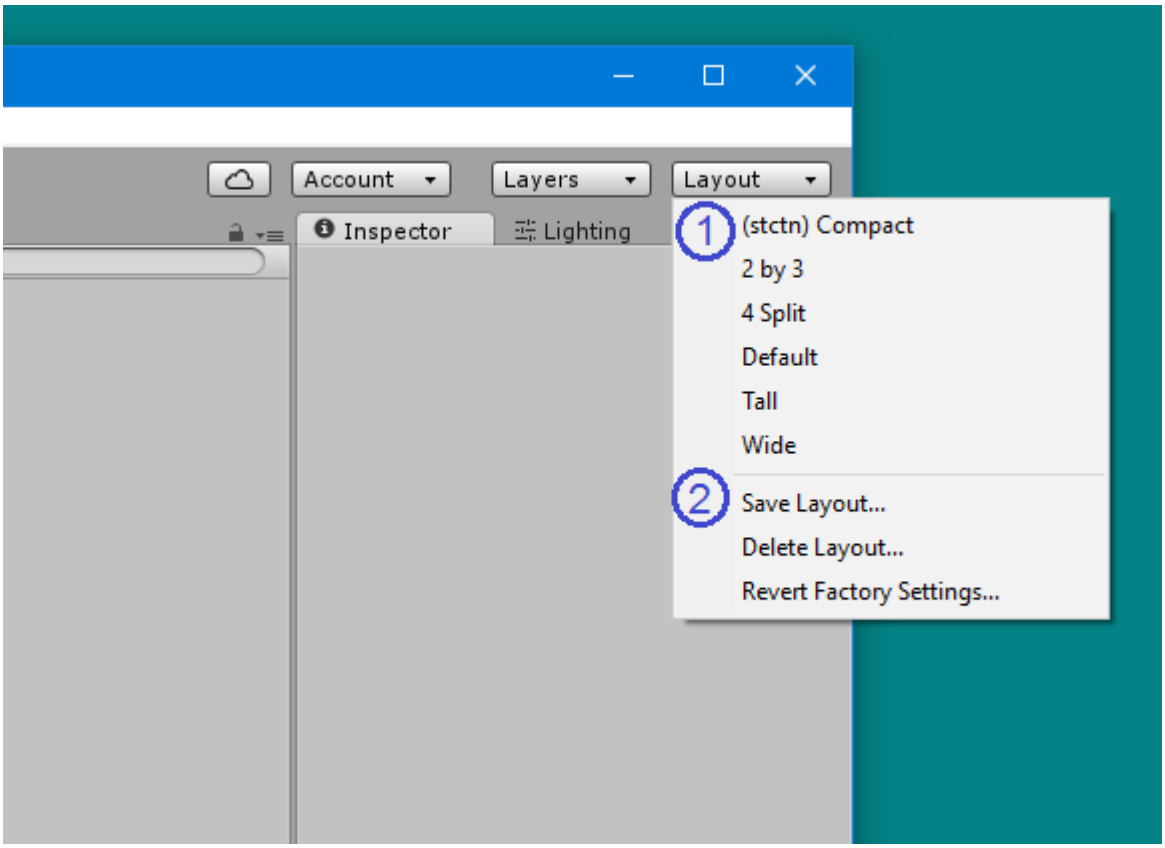
Result should be like below in the Console window:



Editor Layouts

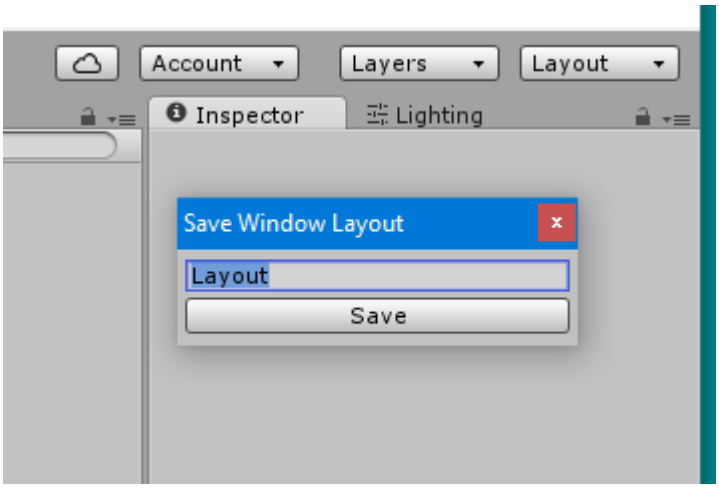
You can save the layout of your tabs and windows to standardize your work environment.

The layouts menu can be found in the upper right corner of Unity Editor:

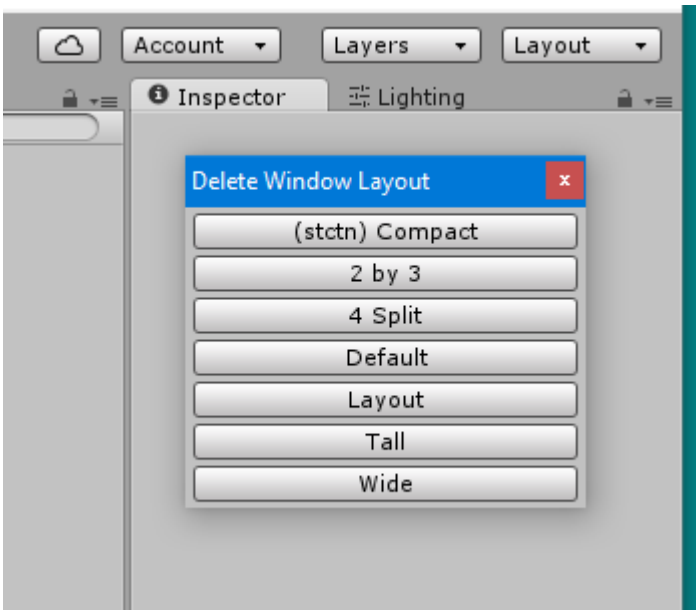


Unity ships with 5 default layouts (2 by 3, 4 Split, Default, Tall, Wide) *(marked with 1)*. In the picture above, aside from default layouts, there is also a custom layout at the top.

You can add your own layouts by clicking **"Save Layout..."** button in the menu *(marked with 2)*:



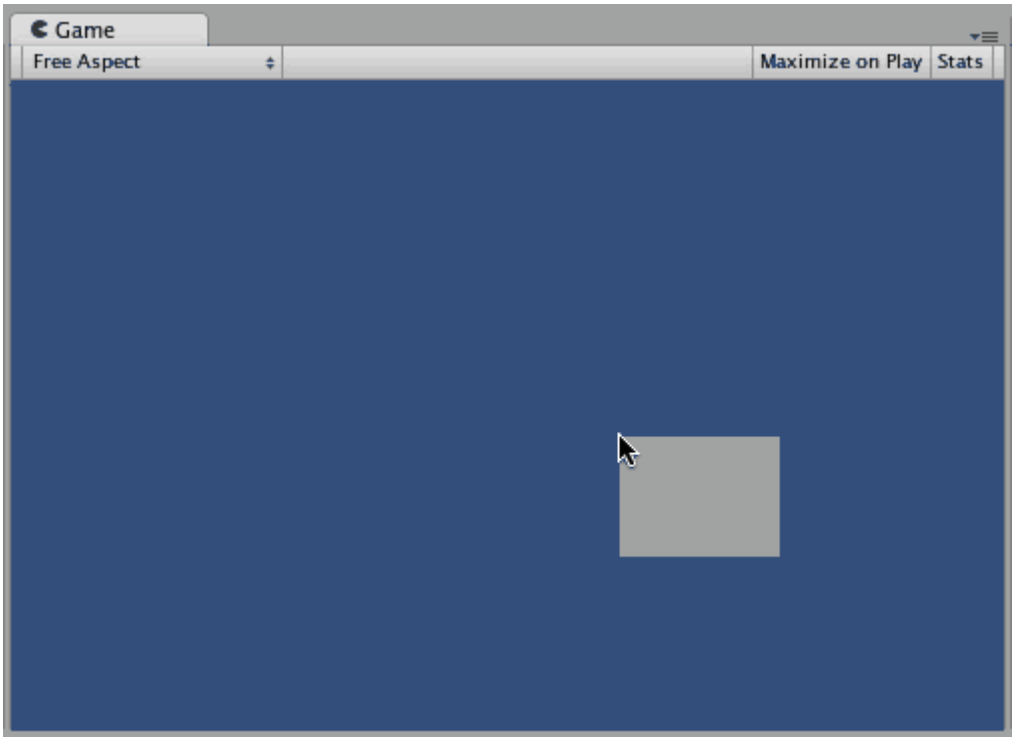
You can also delete any layout by clicking **"Delete Layout..."** button in the menu *(marked with 2)*:



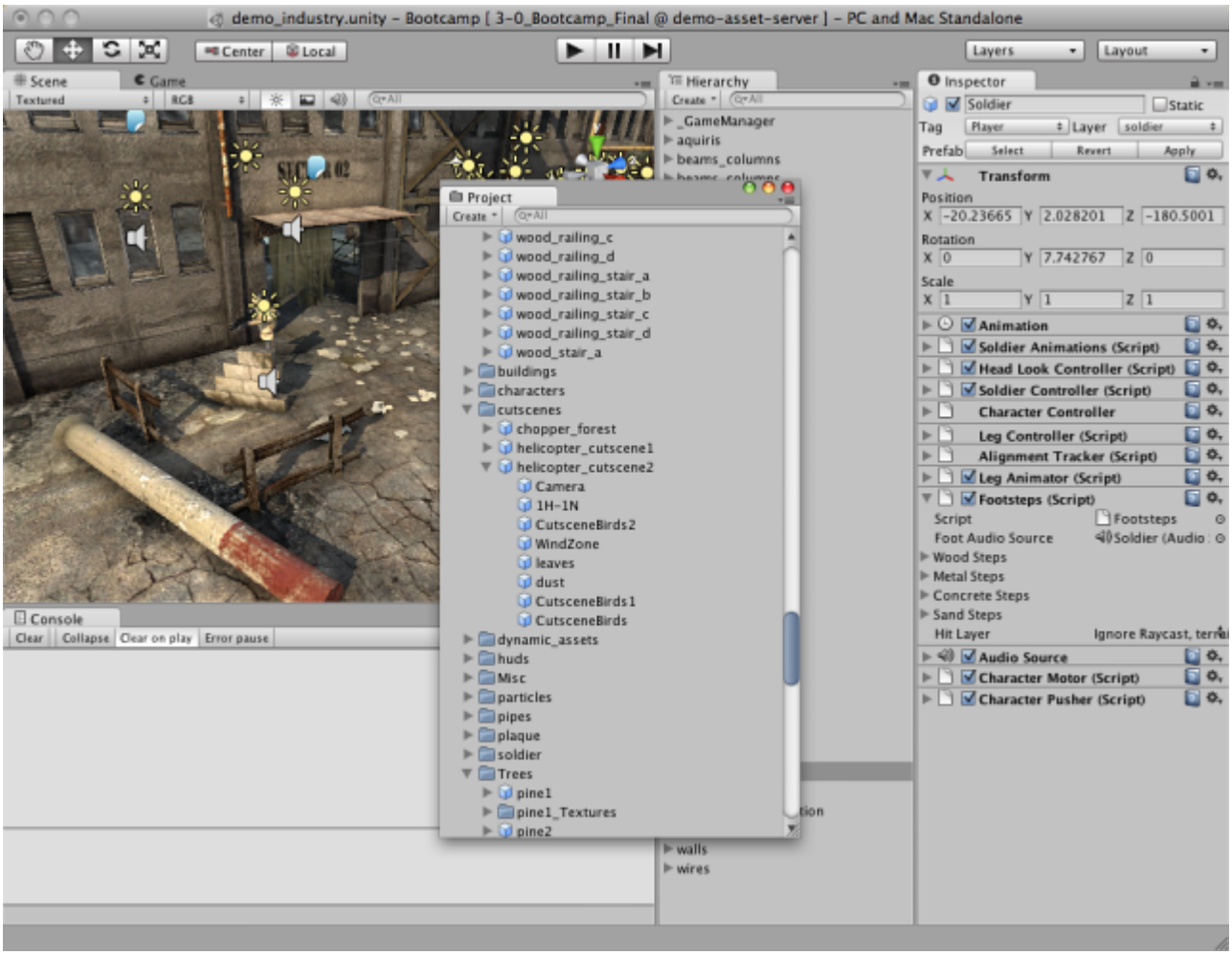
The **"Revert Factory Settings..."** button removes all custom layouts and restores default layouts *(marked with 2)*.

Customizing Your Workspace

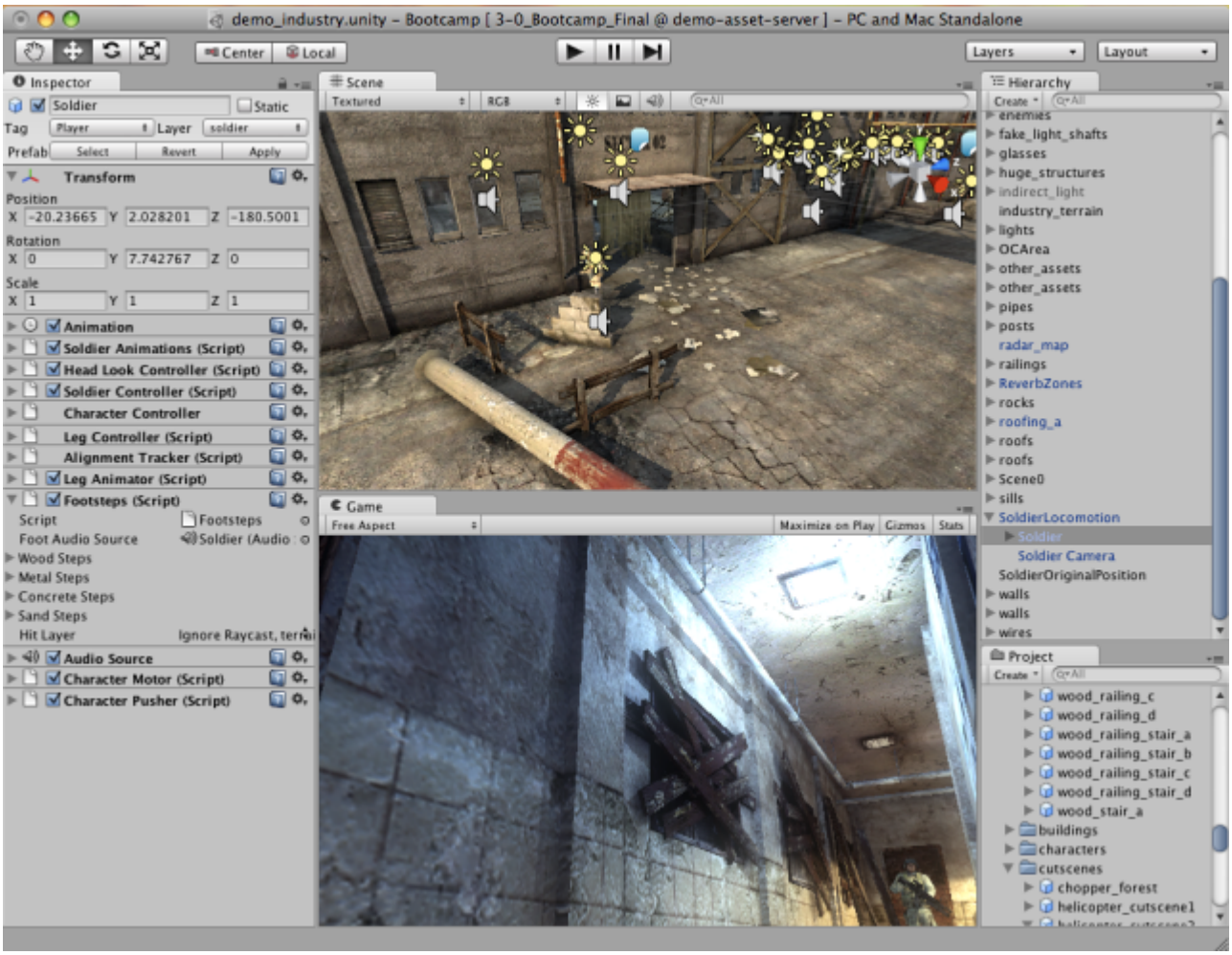
You can customize your Layout of Views by click-dragging the Tab of any View to one of several locations. Dropping a Tab in the Tab Area of an existing window will add the Tab beside any existing Tabs. Alternatively, dropping a Tab in any Dock Zone will add the View in a new window.



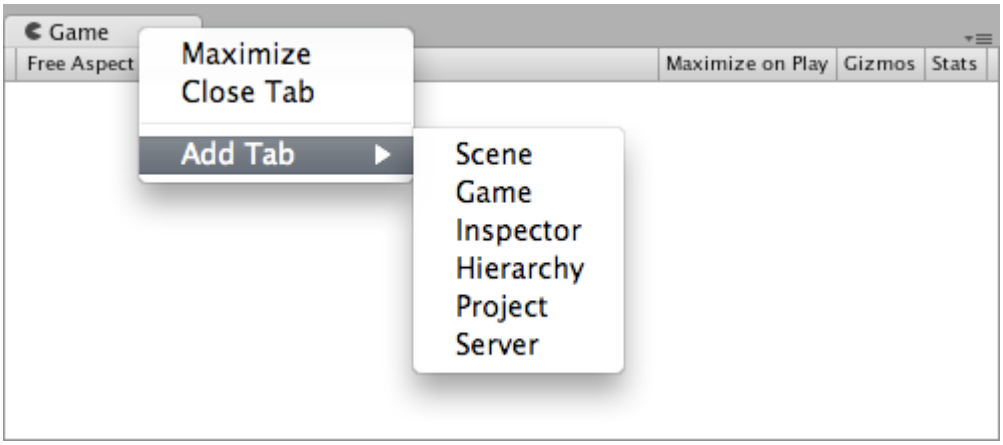
Tabs can also be detached from the Main Editor Window and arranged into their own floating Editor Windows. Floating Windows can contain arrangements of Views and Tabs just like the Main Editor Window.



When you've created an editor layout, you can save the layout and restore it any time. Refer to this example for editor layouts.



At any time, you can right-click the tab of any view to view additional options like Maximize or add a new tab to the same window.



Read Getting started with unity3d online: https://riptutorial.com/unity3d/topic/846/getting-started-with-unity3d

Chapter 2: Ads integration

Introduction

This topic is about the integration of third-party advertisement services, such as Unity Ads or Google AdMob, into a Unity project.

Remarks

This applies to Unity Ads.

> **Make sure that Test Mode for Unity Ads is enabled during development**

You, as the developer, are not allowed to generate impressions or installs by clicking on ads in your own game. Doing so violates the Unity Ads Terms of Service agreement, and you will be banned from the Unity Ads network for attempted fraud.

For more information, read the Unity Ads Terms of Service agreement.

Examples

Unity Ads Basics in C#

```
using UnityEngine;
using UnityEngine.Advertisements;

public class Example : MonoBehaviour
{
    #if !UNITY_ADS // If the Ads service is not enabled
    public string gameId; // Set this value from the inspector
    public bool enableTestMode = true; // Enable this during development
    #endif

    void InitializeAds () // Example of how to initialize the Unity Ads service
    {
        #if !UNITY_ADS // If the Ads service is not enabled
        if (Advertisement.isSupported) { // If runtime platform is supported
            Advertisement.Initialize(gameId, enableTestMode); // Initialize
        }
        #endif
    }

    void ShowAd () // Example of how to show an ad
    {
        if (Advertisement.isInitialized || Advertisement.IsReady()) { // If the ads are ready
to be shown
            Advertisement.Show(); // Show the default ad placement
        }
    }
}
```

Unity Ads Basics in JavaScript

```
#pragma strict
import UnityEngine.Advertisements;

#if !UNITY_ADS // If the Ads service is not enabled
public var gameId : String; // Set this value from the inspector
public var enableTestMode : boolean = true; // Enable this during development
#endif

function InitializeAds () // Example of how to initialize the Unity Ads service
{
    #if !UNITY_ADS // If the Ads service is not enabled
    if (Advertisement.isSupported) { // If runtime platform is supported
        Advertisement.Initialize(gameId, enableTestMode); // Initialize
    }
    #endif
}

function ShowAd () // Example of how to show an ad
{
    if (Advertisement.isInitialized && Advertisement.IsReady()) { // If the ads are ready to
be shown
        Advertisement.Show(); // Show the default ad placement
    }
}
```

Read Ads integration online: https://riptutorial.com/unity3d/topic/9796/ads-integration

Chapter 3: Android Plugins 101 - An Introduction

Introduction

This topic is the first part of a series on how to create Android Plugins for Unity. Start here if you have little to no experience creating plugins, and/or the Android OS.

Remarks

Through this series, I extensively use external links that I encourage you to read. While paraphrased versions of the relevant content will be included here, there may be times when the additional reading will help.

Beginning with Android plugins

Currently, Unity provides two ways to call native Android code.

1. Write native Android code in Java, and call these Java functions using C#
2. Write C# code to directly call functions that are part of the Android OS

To interact with native code, Unity provides some classes and functions.

- AndroidJavaObject - This is the base class that Unity provides to interact with native code. Almost any object returned from native code can be stored as and AndroidJavaObject
- AndroidJavaClass - Inherits from AndroidJavaObject. This is used to reference classes in your native code
- Get / Set values of an instance of a native object, and the static GetStatic / SetStatic versions
- Call / CallStatic to call native non-static & static functions

---

Outline to creating a plugin and terminology

1. Write native Java code in Android Studio
2. Export the code in a JAR / AAR file (Steps here for JAR files and AAR files)
3. Copy the JAR / AAR file into your Unity project at **Assets/Plugins/Android**
4. Write code in Unity (C# has always been the way to go here) to call functions in the plugin

Note that the first three steps apply ONLY if you wish to have a native plugin!

From here on out, I'll refer to the JAR / AAR file as the **native plugin**, and the C# script as the **C#**

Choosing between the plugin creation methods

It's immediately obvious that the first way of creating plugins is long drawn, so choosing your route seems moot. However, method 1 is the ONLY way to call custom code. So, how does one choose?

Simply put, does your plugin

1. Involve custom code - Choose method 1
2. Only invoke native Android functions? - Choose method 2

Please do **NOT** try to "mix" (i.e. a part of the plugin using method 1, and the other using method 2) the two methods! While entirely possible, it's often impractical and painful to manage.

Examples

UnityAndroidPlugin.cs

Create a new C# script in Unity and replace it's contents with the following

```
using UnityEngine;
using System.Collections;

public static class UnityAndroidPlugin {

}
```

UnityAndroidNative.java

Create a new Java class in Android Studio and replace it's contents with the following

```
package com.axs.unityandroidplugin;
import android.util.Log;
import android.widget.Toast;
import android.app.ActivityManager;
import android.content.Context;


public class UnityAndroidNative {


}
```

UnityAndroidPluginGUI.cs

Create a new C# script in Unity and paste these contents

```
using UnityEngine;
using System.Collections;

public class UnityAndroidPluginGUI : MonoBehaviour {

    void OnGUI () {

    }

}
```

Read Android Plugins 101 - An Introduction online:
https://riptutorial.com/unity3d/topic/10032/android-plugins-101---an-introduction

Chapter 4: Asset Store

Examples

Accessing the Asset Store

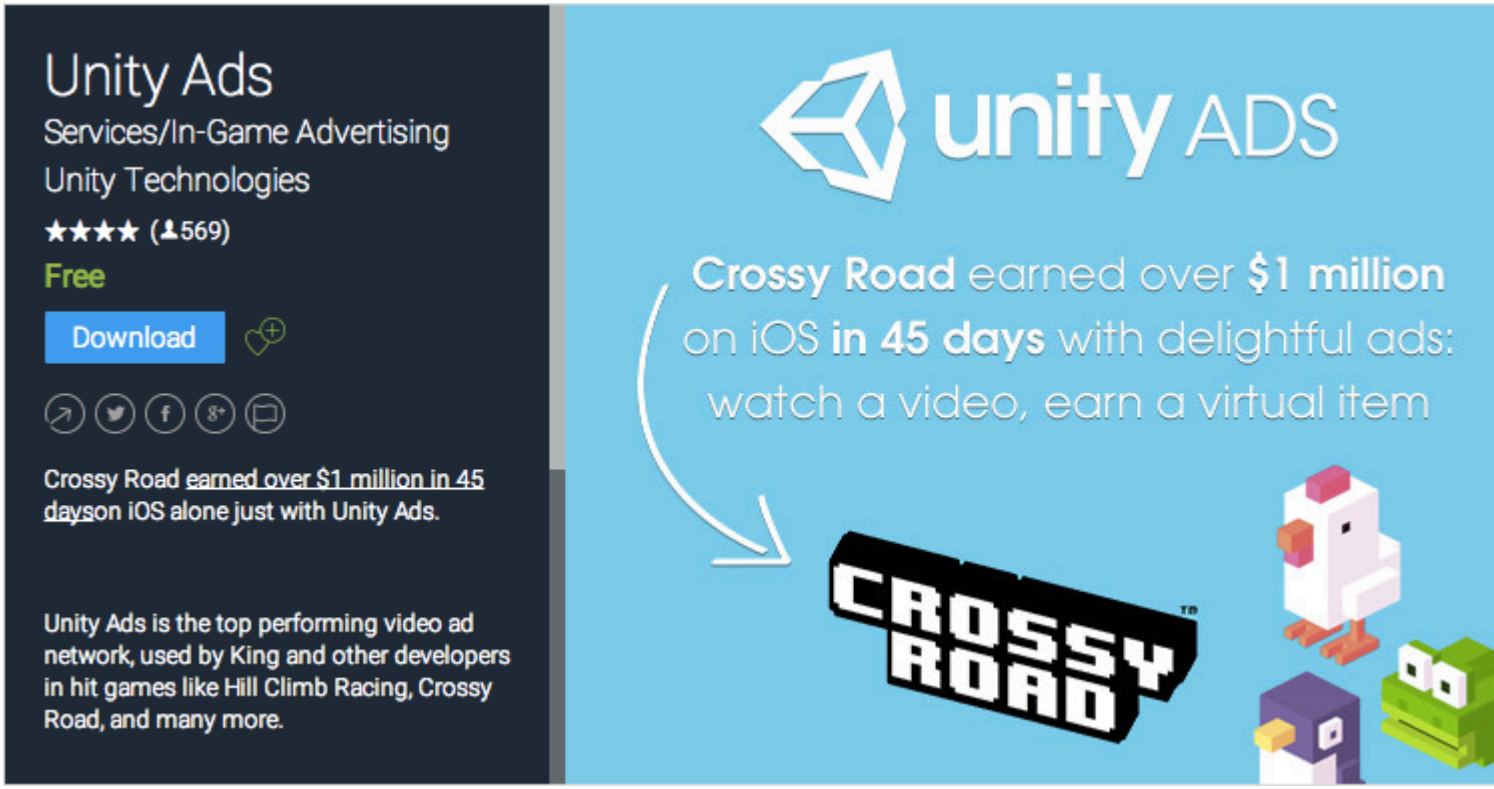
There are three ways you can access the Unity Asset Store:

- Open the Asset Store window by selecting Window→Asset Store from the main menu within Unity.
- Use the Shortcut key (Ctrl+9 on Windows / 9 on Mac OS)
- Browse the web interface: https://www.assetstore.unity3d.com/

You may be prompted to create a free user account or sign-in if it is your first time accessing the Unity Asset Store.

Purchasing Assets

After accessing the Asset Store and viewing the asset you'd like to download, simply click the **Download** button. The button text may also be **Buy Now** if the asset has an associated cost.



If you are viewing the Unity Asset Store through the web interface, the **Download** button text may instead display as **Open in Unity**. Selecting this button will launch an instance of Unity and display the asset within the *Asset Store window*.

You may be prompted to create a free user account or sign-in if it is your first time purchasing from the Unity Asset Store.

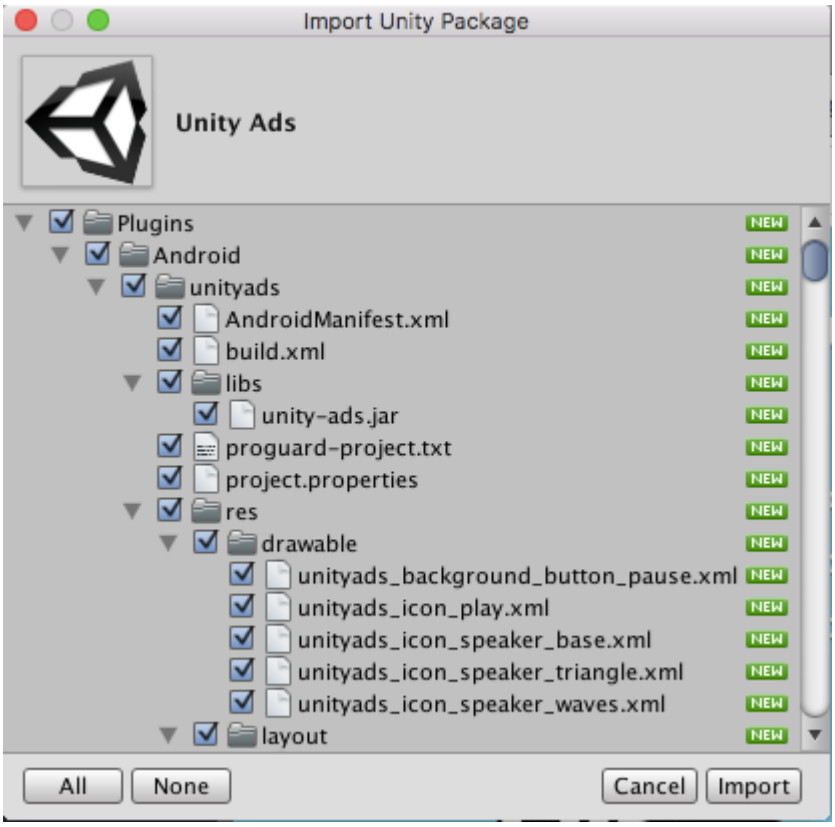
Unity will then proceed with accepting your payment, if applicable.

Importing Assets

After the asset has been downloaded in Unity, the **Download** or **Buy Now** button will change to **Import**.

Selecting this option will prompt the user with a *Import Unity Package* window, where the user may select the asset files of which they'd like to import within their project.

Select **Import** to confirm the process, placing the selected asset files inside the Assets folder shown in the *Project View window*.



Publishing Assets

1. make a publisher account
2. add an asset in the publisher account
3. download the asset store tools (from the asset store)
4. go to "Asset Store Tools" > "Package Upload"
5. select the correct package and project folder in the asset store tools window
6. click upload
7. submit your asset online

TODO - add pictures, more details

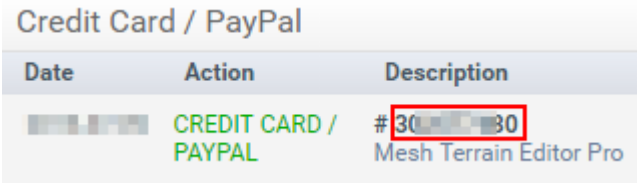
Confirm the invoice number of one purchase

The invoice number is used to verify the sale for publishers. Many publishers of paid asset or plugin ask for the invoice number upon request of support. The invoice number is also used as a license key to activate some asset or plugin.

The invoice number can be found in two place:

1. After you bought the asset, you will be sent an email whose subject is "Unity Asset Store purchase confirmation...". The invoice number is in the PDF attachment of this email.

2. Open https://www.assetstore.unity3d.com/#!/account/transactions, then you can find the invoice number in the *Description* column.



Read Asset Store online: https://riptutorial.com/unity3d/topic/5705/asset-store

Chapter 5: Attributes

Syntax

- [AddComponentMenu(string menuName)]
- [AddComponentMenu(string menuName, int order)]
- [CanEditMultipleObjects]
- [ContextMenuItem(string name, string function)]
- [ContextMenu(string name)]
- [CustomEditor(Type inspectedType)]
- [CustomEditor(Type inspectedType, bool editorForChildClasses)]
- [CustomPropertyDrawer(Type type)]
- [CustomPropertyDrawer(Type type, bool useForChildren)]
- [DisallowMultipleComponent]
- [DrawGizmo(GizmoType gizmo)]
- [DrawGizmo(GizmoType gizmo, Type drawnGizmoType)]
- [ExecuteInEditMode]
- [Header(string header)]
- [HideInInspector]
- [InitializeOnLoad]
- [InitializeOnLoadMethod]
- [MenuItem(string itemName)]
- [MenuItem(string itemName, bool isValidateFunction)]
- [MenuItem(string itemName, bool isValidateFunction, int priority)]
- [Multiline(int lines)]
- [PreferenceItem(string name)]
- [Range(float min, float max)]
- [RequireComponent(Type type)]
- [RuntimeInitializeOnLoadMethod]
- [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType loadType)]
- [SerializeField]
- [Space(float height)]
- [TextArea(int minLines, int maxLines)]
- [Tooltip(string tooltip)]

Remarks

SerializeField

Unity's serialization system can be used to do the following:

- **Can** serialize public nonstatic fields (of serializable types)
- **Can** serialize nonpublic nonstatic fields marked with the [SerializeField] attribute

- **Cannot** serialize static fields
- **Cannot** serialize static properties

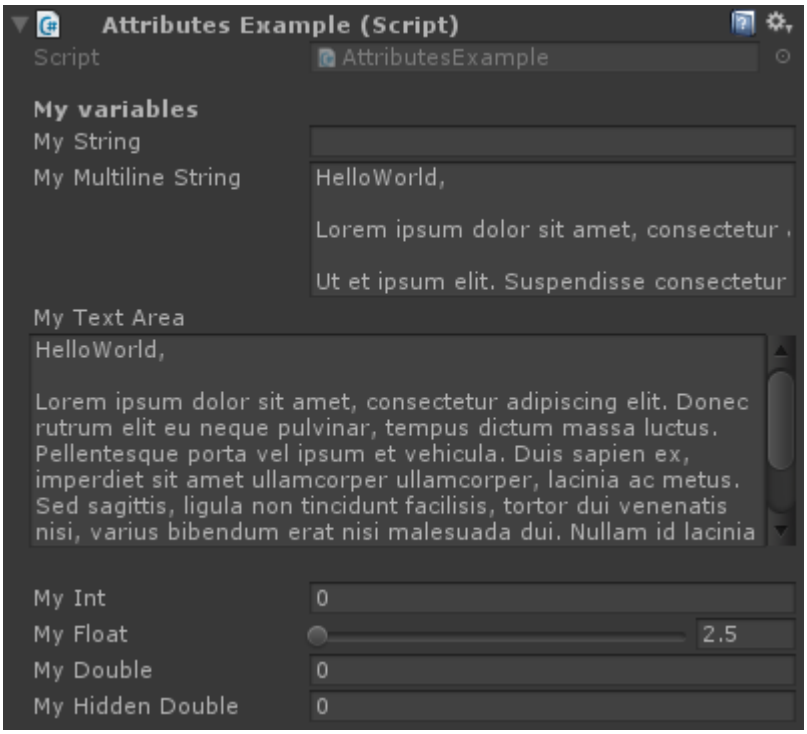
Your field, even if marked with the SerializeField attribute, will only be attributed if it is of a type that Unity can serialize, which are:

- All classes inheriting from UnityEngine.Object (e.g. GameObject, Component, MonoBehaviour, Texture2D)
- All basic data types like int, string, float, bool
- Some built-in types like Vector2/3/4, Quaternion, Matrix4x4, Color, Rect, LayerMask
- Arrays of a serializable type
- List of a serializable type
- Enums
- Structs

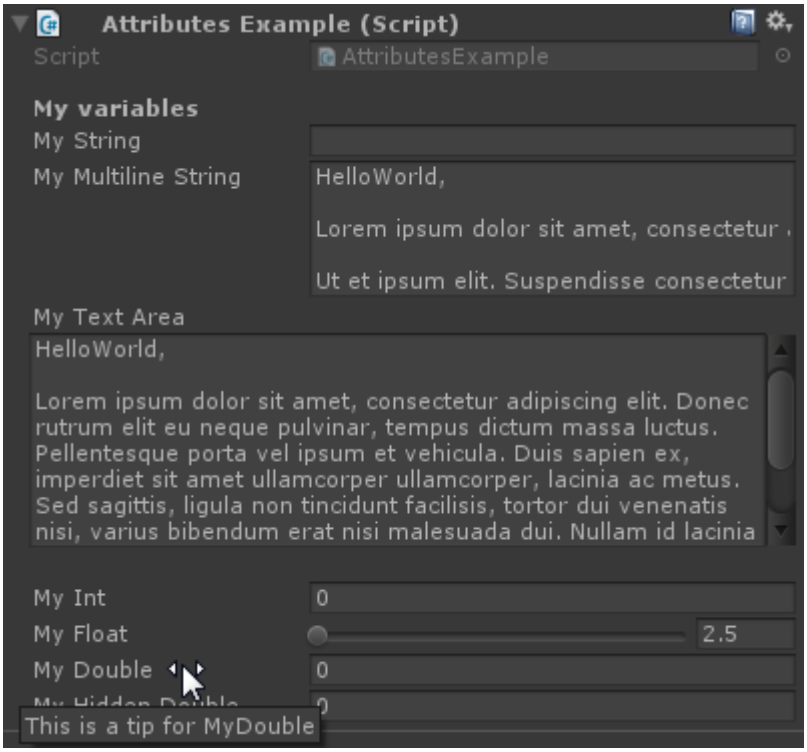
Examples

Common inspector attributes

```
[Header( "My variables" )]
public string MyString;

[HideInInspector]
public string MyHiddenString;

[Multiline( 5 )]
public string MyMultilineString;

[TextArea( 2, 8 )]
public string MyTextArea;

[Space( 15 )]
public int MyInt;

[Range( 2.5f, 12.5f )]
public float MyFloat;

[Tooltip( "This is a tip for MyDouble" )]
public double MyDouble;

[SerializeField]
private double myHiddenDouble;
```



When hovering over the label of a field:



```
[Header( "My variables" )]
public string MyString;
```

Header places a bold label containing the text above the attributed field. This is often used for labeling groups to make them stand out against other labels.

```
[HideInInspector]
public string MyHiddenString;
```

HideInInspector prevents public fieldsfrom being shown in the inspector. This is useful for accessing fields from other parts of code where they aren't otherwise visible or mutable.

```
[Multiline( 5 )]
public string MyMultilineString;
```

Multiline creates a textbox with a specified number of lines. Exceeding this amount will neither expand the box nor wrap the text.

```
[TextArea( 2, 8 )]
public string MyTextArea;
```

TextArea allows multiline-style text with automatic word-wrapping and scroll bars if the text exceeds the allotted area.

```
[Space( 15 )]
public int MyInt;
```

Space forces the inspector to add extra space between previous and current items -useful in distinguishing and separating groups.

```
[Range( 2.5f, 12.5f )]
public float MyFloat;
```

Range forces a numerical value between a minimum and a maximum. This attribute also works on integers and doubles, even though min and max are specified as floats.

```
[Tooltip( "This is a tip for MyDouble" )]
public double MyDouble;
```

Tooltip shows an additional description whenever the field's label is hovered over.

```
[SerializeField]
private double myHiddenDouble;
```

SerializeField forces Unity to serialize the field - useful for private fields.

Component attributes

```
[DisallowMultipleComponent]
[RequireComponent( typeof( Rigidbody ) )]
public class AttributesExample : MonoBehaviour
{
    [...]
}
```

```
[DisallowMultipleComponent]
```

The DisallowMultipleComponent attribute prevents users adding multiple instances of this component to one GameObject.

```
[RequireComponent( typeof( Rigidbody ) )]
```

The RequireComponent attribute allows you to specify another component (or more) as requirements for when this component is added to a GameObject. When you add this component to a GameObject, the required components will be automatically added (if not already present) and those components cannot be removed until the one that requires them is removed.

Runtime attributes

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
{

    [RuntimeInitializeOnLoadMethod]
    private static void FooBar()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
    private static void Foo()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
    private static void Bar()
    {
        [...]
    }

    void Update()
    {
        if ( Application.isEditor )
        {
            [...]
        }
        else
        {
            [...]
        }
    }
}
```

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
```

The ExecuteInEditMode attribute forces Unity to execute this script's magic methods even while the game is not playing.

> The functions are not constantly called like in play mode

- Update is only called when something in the scene changed.
- OnGUI is called when the Game View receives an Event.
- OnRenderObject and the other rendering callback functions are called on every repaint of the Scene View or Game View.

```
[RuntimeInitializeOnLoadMethod]
private static void FooBar()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
private static void Foo()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
private static void Bar()
```

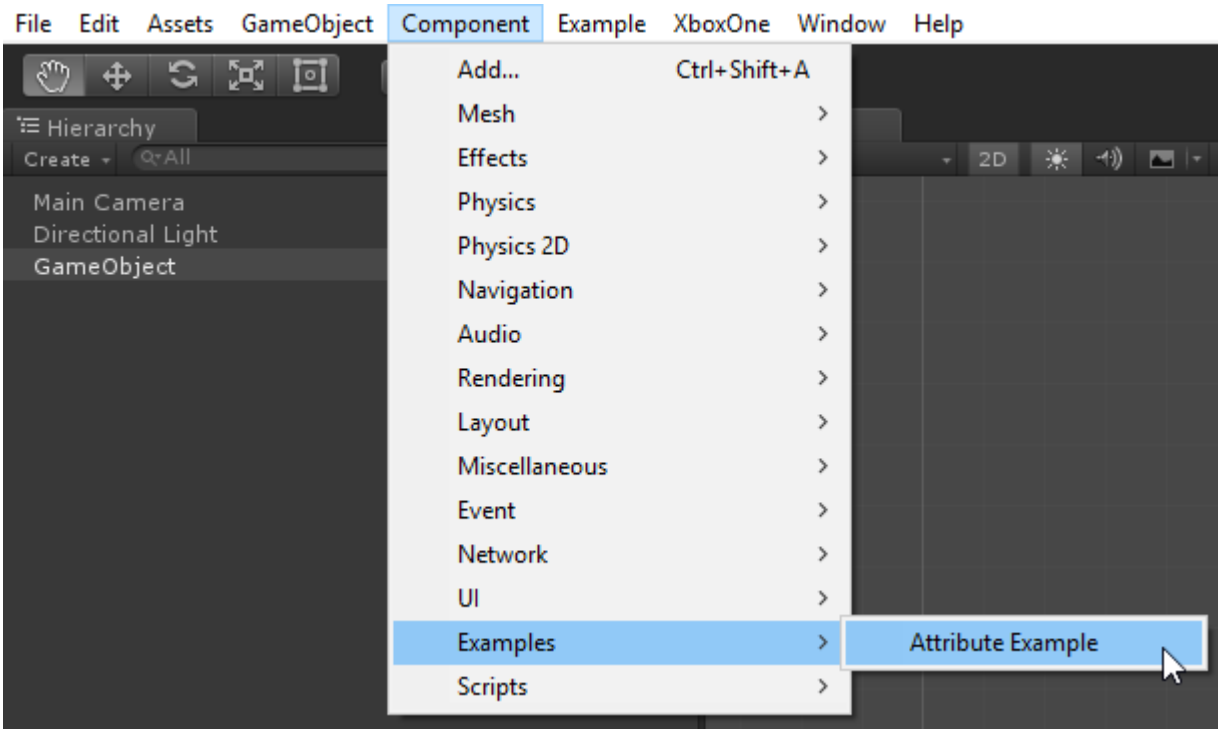
The RuntimeInitializeOnLoadMethod attribute allows a runtime class method to be called when the game loads the runtime, without any interaction from the user.

You can specify if you want the method to be invoked before or after scene load (after is default). The order of execution is not guaranteed for methods using this attribute.

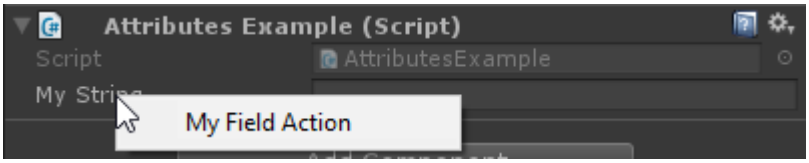
Menu attributes

```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
{
    [ContextMenuItem( "My Field Action", "MyFieldContextAction" )]
    public string MyString;

    private void MyFieldContextAction()
    {
        [...]
    }

    [ContextMenu( "My Action" )]
    private void MyContextMenuAction()
    {
        [...]
    }
}
```

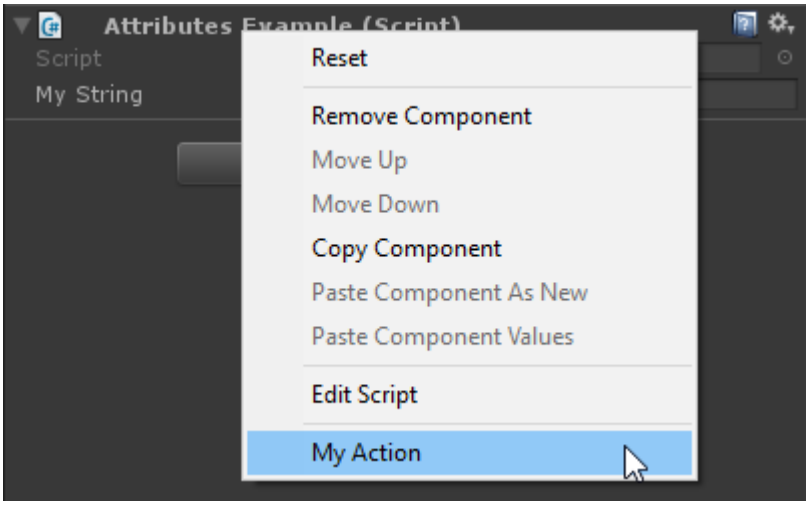
The result of the [AddComponentMenu] attribute



The result of the [ContextMenuItem] attribute



The result of the [ContextMenu] attribute



```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
```

The AddComponentMenu attribute allows you to place your component anywhere in the Component menu instead of the Component->Scripts menu.

```
[ContextMenuItem( "My Field Action", "MyFieldContextAction" )]
public string MyString;
```

```
private void MyFieldContextAction()
{
    [...]
}
```

The ContextMenuItem attribute allows you to define functions that can be added to the context menu of a field. These functions will be executed upon selection.

```
[ContextMenu( "My Action" )]
private void MyContextMenuAction()
{
    [...]
}
```

The ContextMenu attribute allows you to define functions that can be added to the context menu of the component.

Editor attributes

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{

    static AttributesExample()
    {
        [...]
    }

    [InitializeOnLoadMethod]
    private static void Foo()
    {
        [...]
    }
}
```

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{

    static AttributesExample()
    {
        [...]
    }
```

The InitializeOnLoad attribute allows the user to initialize a class without any interaction from the user. This happens whenever the editor launches or on a recompile. The static constructor guarantees that this will be called before any other static functions.

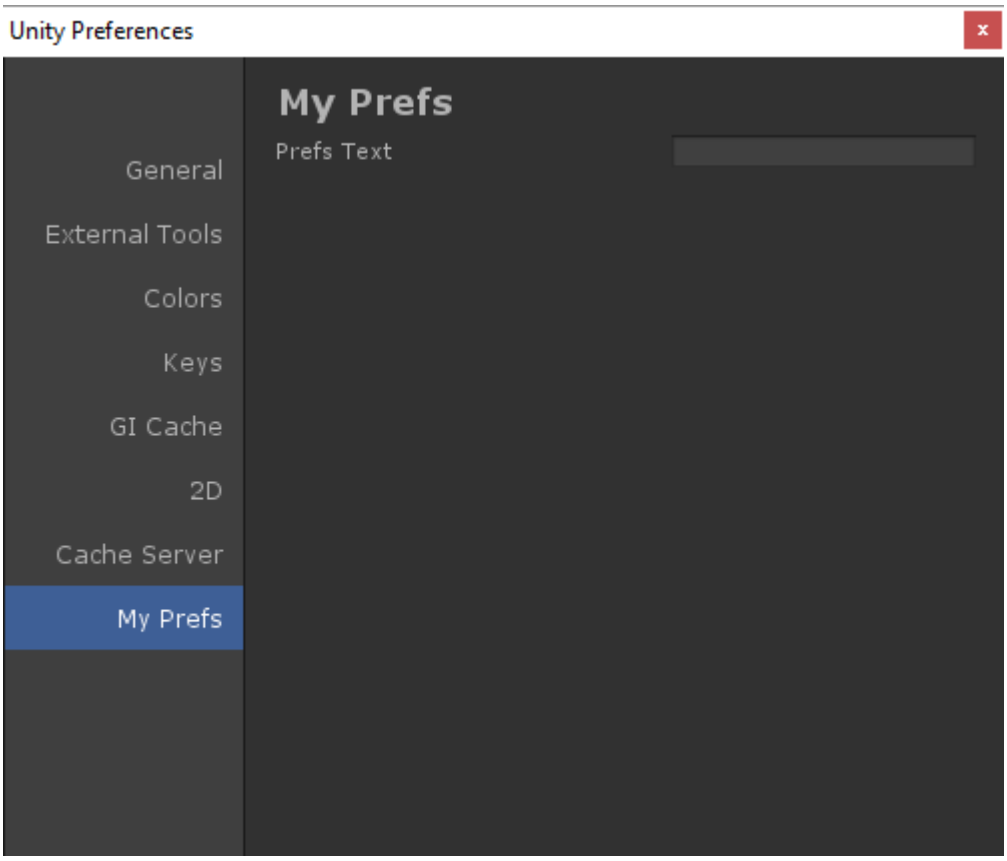
```
[InitializeOnLoadMethod]
private static void Foo()
{
    [...]
```

```
}
```

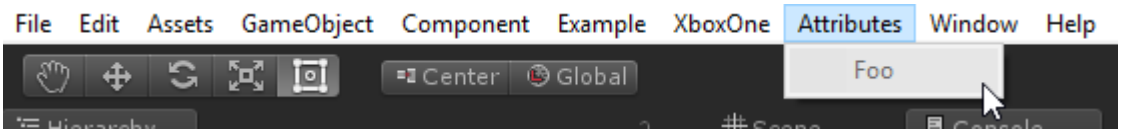
The InitializeOnLoad attribute allows the user to initialize a class without any interaction from the user. This happens whenever the editor launches or on a recompile. The order of execution is not guaranteed for methods using this attribute.

```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
{

    public int MyInt;

    private static string prefsText = "";

    [PreferenceItem( "My Prefs" )]
    public static void PreferencesGUI()
    {
        prefsText = EditorGUILayout.TextField( "Prefs Text", prefsText );
    }

    [MenuItem( "Attributes/Foo" )]
    private static void Foo()
    {
        [...]
    }

    [MenuItem( "Attributes/Foo", true )]
    private static bool FooValidate()
    {
        return false;
    }
}
```

The result of the [PreferenceItem] attribute



The result of the [MenuItem] attribute



```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
```

The CanEditMultipleObjects attribute allows you to edit values from your component over multiple GameObjects. Without this component you won't see your component appear like normal when selecting multiple GameObjects but instead you will see the message "Multi-object editing not supported"

> This attribute is for custom editors to support multi editing. Non-custom editors automatically support multi editing.

```
[PreferenceItem( "My Prefs" )]
public static void PreferencesGUI()
```

The PreferenceItem attribute allows to you create an extra item in Unity's preferences menu. The receiving method needs to be static for it to be used.

```
[MenuItem( "Attributes/Foo" )]
private static void Foo()
{
    [...]
```

```
}

[MenuItem( "Attributes/Foo", true )]
private static bool FooValidate()
{
    return false;
}
```

The MenuItem attribute allows you to create custom menu items to execute functions. This example uses a validator function as well (which always returns false) to prevent execution of the function.

```
[CustomEditor( typeof( MyComponent ) )]
public class AttributesExample : Editor
{
    [...]
}
```

The CustomEditor attribute allows you to create custom editors for your components. These editors will be used for drawing your component in the inspector and need to derive from the Editor class.

```
[CustomPropertyDrawer( typeof( MyClass ) )]
public class AttributesExample : PropertyDrawer
{
    [...]
}
```

The CustomPropertyDrawer attribute allows you to create a custom property drawer for in the inspector. You can use these drawers for your custom data types so that they can be seen used in the inspector.

```
[DrawGizmo( GizmoType.Selected )]
private static void DoGizmo( AttributesExample obj, GizmoType type )
{
    [...]
}
```

The DrawGizmo attribute allows you to draw custom gizmos for your components. These gizmos will be drawn in the Scene View. You can decide when to draw the gizmo by using the GizmoType parameter in the DrawGizmo attribute.

> The receiving method requires two parameters, the first is the component to draw the gizmo for and the second is the state that the object who needs the gizmo drawn is in.

Read Attributes online: https://riptutorial.com/unity3d/topic/5535/attributes

Chapter 6: Audio System

Introduction

This is a documentation about playing audio in Unity3D.

Examples

Audio class - Play audio

```
using UnityEngine;

public class Audio : MonoBehaviour {
    AudioSource audioSource;
    AudioClip audioClip;

    void Start() {
        audioClip = (AudioClip)Resources.Load("Audio/Soundtrack");
        audioSource.clip = audioClip;
        if (!audioSource.isPlaying) audioSource.Play();
    }
```

Read Audio System online: https://riptutorial.com/unity3d/topic/8064/audio-system

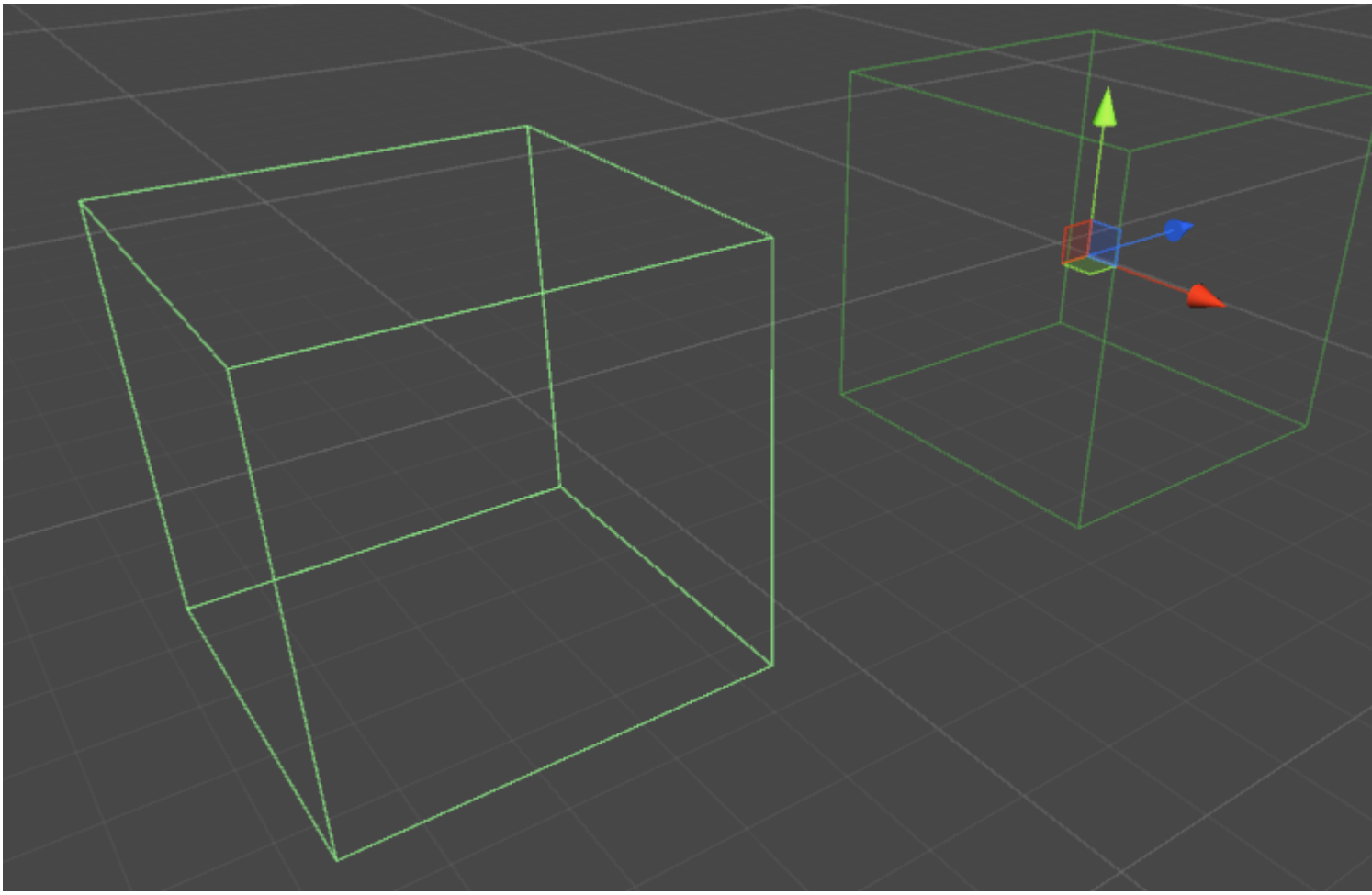
Chapter 7: Collision

Examples

Colliders

Box Collider

A primitive Collider shaped like a cuboid.



Properties

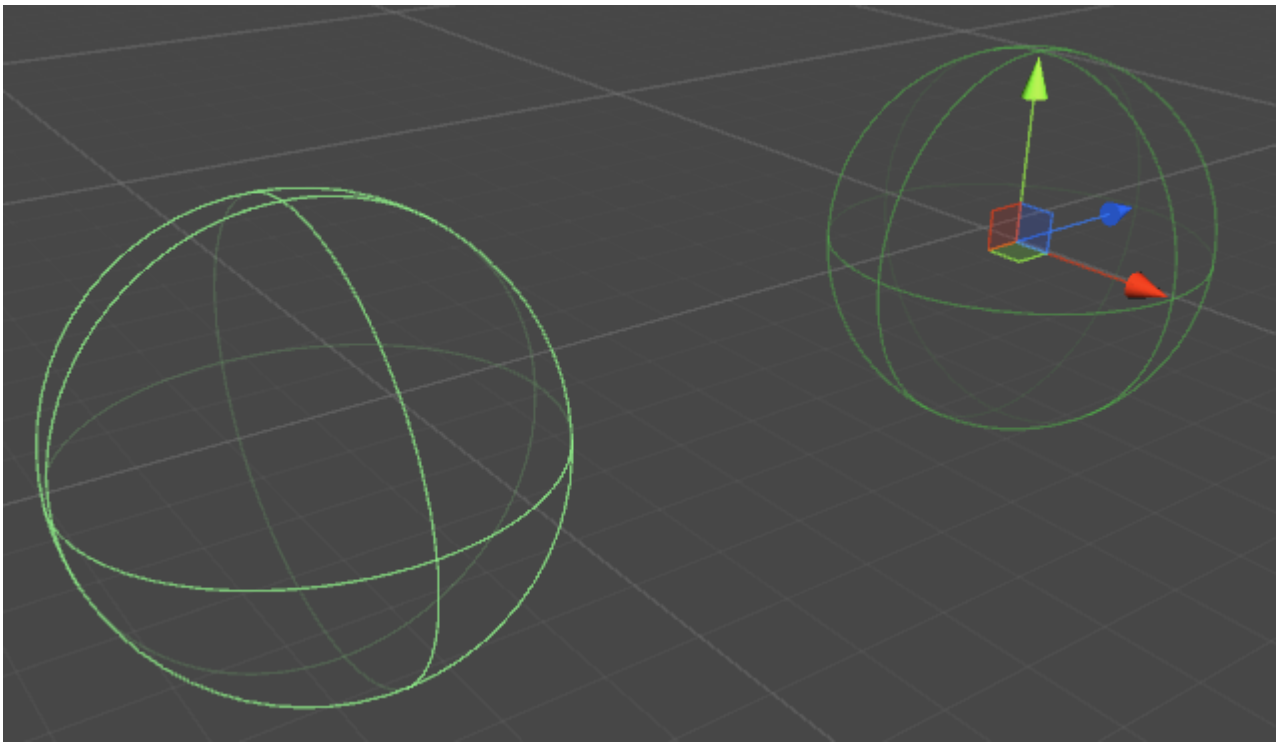
- **Is Trigger** - If ticked, the Box Collider will ignore physics and become a Trigger Collider
- **Material** - A reference, if specified, to the physics material of the Box Collider
- **Center** - The Box Collider's central position in local space
- **Size** - The size of the Box Collider measured in local space

Example

```
// Add a Box Collider to the current GameObject.
BoxCollider myBC = BoxCollider)myGameObject.gameObject.AddComponent(typeof(BoxCollider));

// Make the Box Collider into a Trigger Collider.
myBC.isTrigger= true;

// Set the center of the Box Collider to the center of the GameObject.
myBC.center = Vector3.zero;

// Make the Box Collider twice as large.
myBC.size = 2;
```

Sphere Collider

A primitive Collider shaped like a sphere.



Properties

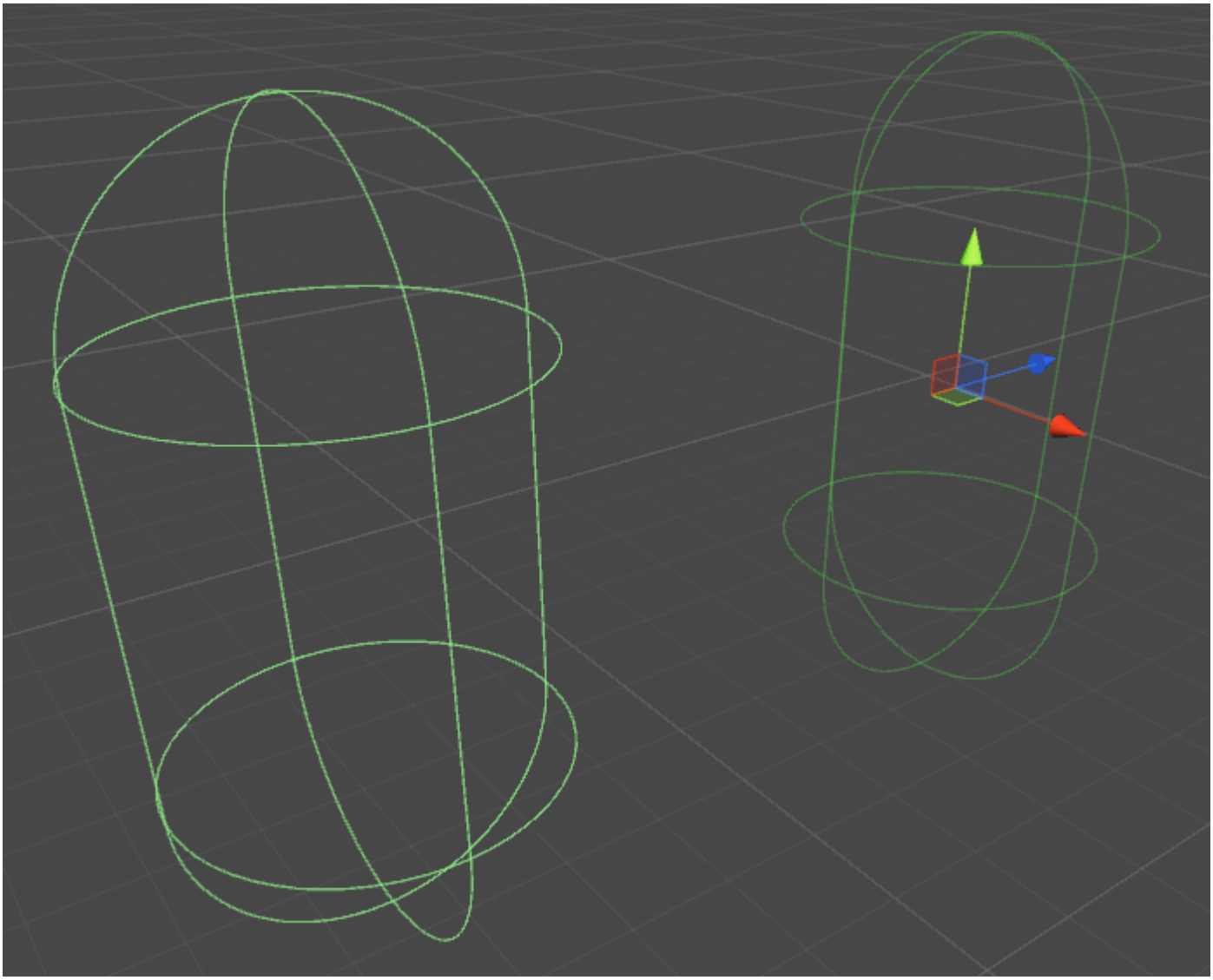
- **Is Trigger** - If ticked, the Sphere Collider will ignore physics and become a Trigger Collider
- **Material** - A reference, if specified, to the physics material of the Sphere Collider
- **Center** - The Sphere Collider's central position in local space
- **Radius** - The radius of the Collider

Example

```
// Add a Sphere Collider to the current GameObject.
SphereCollider mySC =
SphereCollider)myGameObject.gameObject.AddComponent(typeof(SphereCollider));

// Make the Sphere Collider into a Trigger Collider.
mySC.isTrigger= true;

// Set the center of the Sphere Collider to the center of the GameObject.
mySC.center = Vector3.zero;

// Make the Sphere Collider twice as large.
mySC.radius = 2;
```

Capsule Collider

Two half spheres joined by a cylinder.



Properties

- **Is Trigger** - If ticked, the Capsule Collider will ignore physics and become a Trigger Collider
- **Material** - A reference, if specified, to the physics material of the Capsule Collider
- **Center** - The Capsule Collider's central position in local space
- **Radius** - The radius in local space
- **Height** - Total height of the Collider
- **Direction** - The axis of orientation in local space

Example

```
// Add a Capsule Collider to the current GameObject.
CapsuleCollider myCC =
CapsuleCollider)myGameObject.gameObject.AddComponent(typeof(CapsuleCollider));

// Make the Capsule Collider into a Trigger Collider.
myCC.isTrigger= true;

// Set the center of the Capsule Collider to the center of the GameObject.
myCC.center = Vector3.zero;

// Make the Sphere Collider twice as tall.
myCC.height= 2;

// Make the Sphere Collider twice as wide.
myCC.radius= 2;

// Set the axis of lengthwise orientation to the X axis.
myCC.direction = 0;

// Set the axis of lengthwise orientation to the Y axis.
myCC.direction = 1;

// Set the axis of lengthwise orientation to the Y axis.
myCC.direction = 2;
```

Wheel Collider

Properties

- **Mass** - The mass of the Wheel Collider
- **Radius** - The radius in local space
- **Wheel damping rate** - Damping value for the Wheel Collider

- **Suspension distance** - Maximum extension along the Y axis in local space
- **Force app point distance** - The point where forces will be applied,
- **Center** - Center of the Wheel Collider in local space

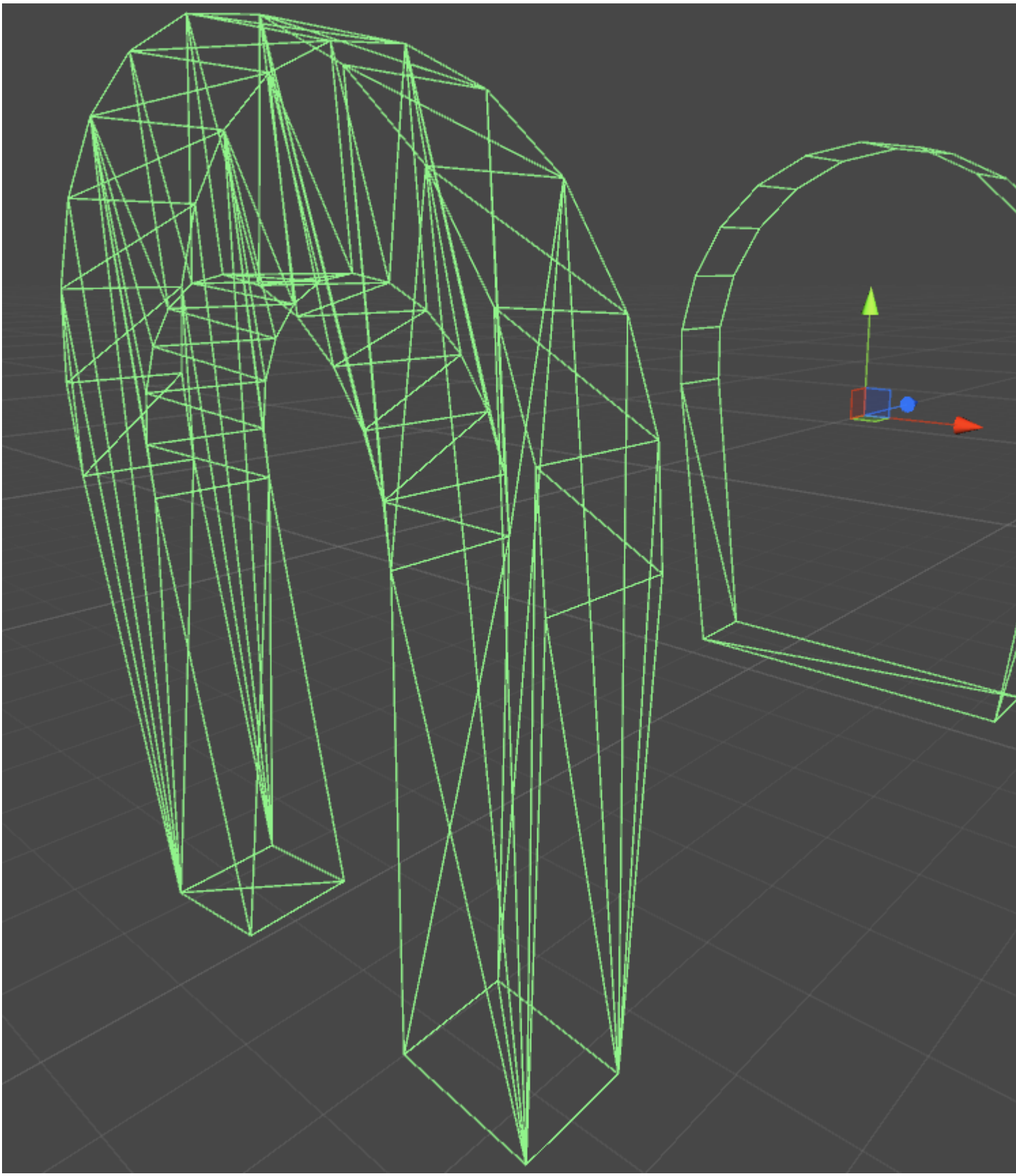
Suspension Spring

- **Spring** - the rate at which the Wheel tries to return to the Target Position
- **Damper** - A larger value dampens the velocity more and the suspension moves slower
- **Target position** - the default is 0.5, at 0 the suspension is bottomed out, at 1 it is at full extension
- **Forward/Sideways friction** - how the tire behaves when rolling forwards or sideways

Example

Mesh Collider

A Collider based on a Mesh Asset.



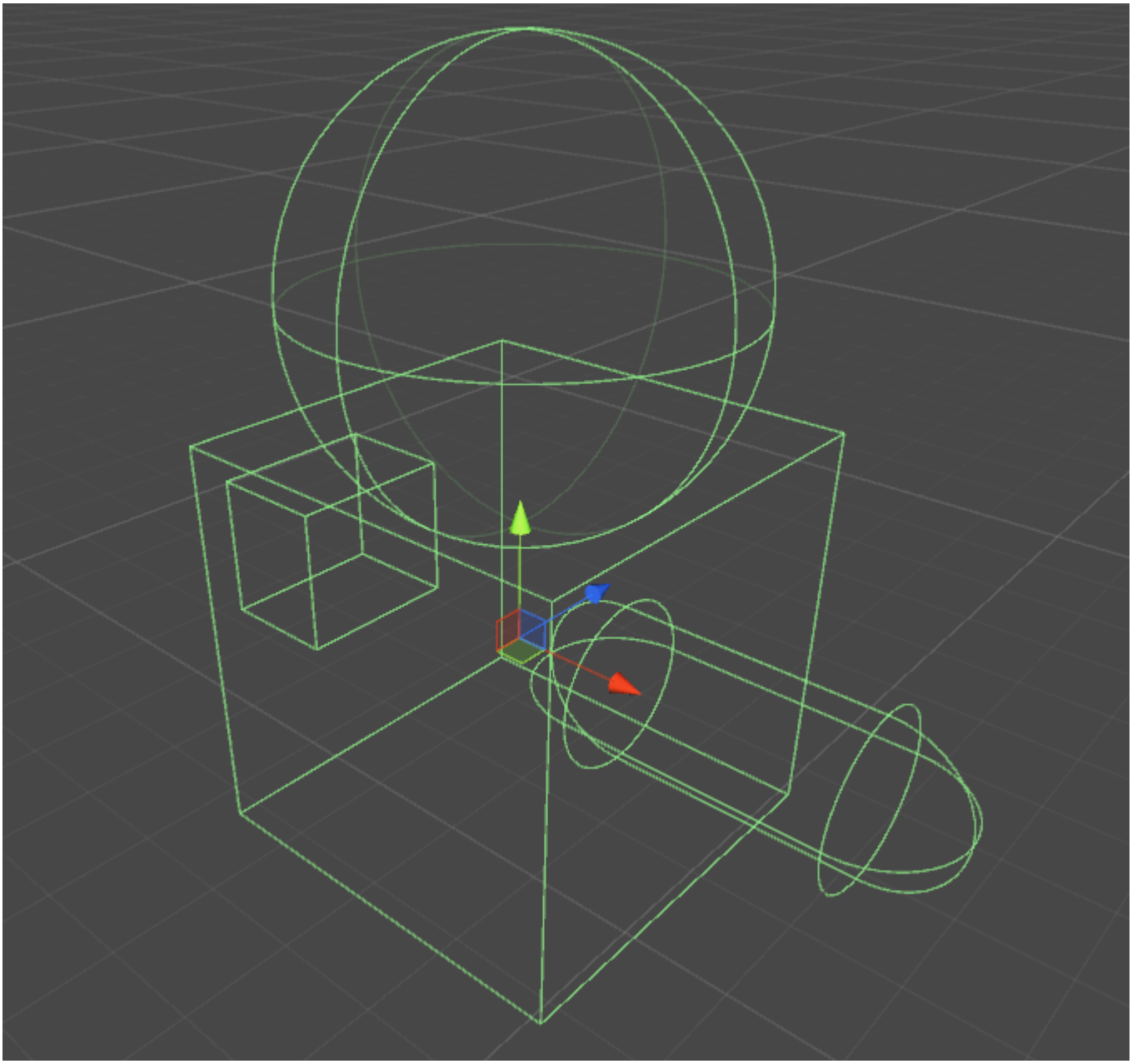
Properties

- **Is Trigger** - If ticked, the Box Collider will ignore physics and become a Trigger Collider
- **Material** - A reference, if specified, to the physics material of the Box Collider

- **Mesh** - A reference to the mesh the Collider is based on
- **Convex** - Convex Mesh colliders are limited to 255 polygons - if enabled, this Collider can collide with other mesh colliders

Example

If you apply more than one Collider to a GameObject, we call it a Compound Collider.



Wheel Collider

The wheel collider inside unity is built upon Nvidia's PhysX wheel collider, and therefore shares many similar properties. Technically unity is a "unitless" program, but to make everything make sense, some standard units are required.

Basic Properties

- Mass - the weight of the wheel in Kilograms, this is used for wheel momentum and the moment of interia when spinning.
- Radius - in meters, the radius of the collider.
- Wheel Damping Rate - Adjusts how "responsive" the wheels are to applied torque.
- Suspension Distance - Total travel distance in meters that the wheel can travel
- Force App Point Distance - where is the force from the suspension applied to the parent rigidbody
- Center - The center position of the wheel

Suspension Settings

- Spring - This is the spring constant, K , in Newtons/meter in the equation:

  Force = Spring Constant * Distance

A good starting point for this value should be the total mass of your vehicle, divided by the number of wheels, multiplied by a number between 50 to 100. E.g. if you have a 2,000kg car with 4 wheels, then each wheel would need to support 500kg. Multiply this by 75, and your spring constant should be 37,500 Newtons/meter.

- Damper - the equivalent of a shock absorber in a car. Higher rates make the suspense "stiffer" and lower rates make it "softer" and more likely to oscillate.
  I do not know the units or equation for this, I think it it has to do with a frequency equation in physics though.

Sideways Friction Settings

The friction curve in unity has a slip value determined by how much the wheel is slipping (in m/s) from the desired position vs. the actual position.

- Extremum Slip - This is the maximum amount (in m/s) a wheel can slip before it should lose traction

- Extremum Value - This is the maximum amount of friction that should be applied to a wheel.

The values for Exrtremum Slip should be between .2 and 2m/s for most realistic cars. 2m/s is about 6 feet per second or 5mph, which is a lot of slip. If you feel that your vehicle needs to have a value higher than 2m/s for slip, you should consider increasing max friction (Extremum Value).

Max Fraction(Extremum Value) is the friction coefficient in the equation:

Force of Friction(in newtons) = Coefficient of Friction * Downward Force(in newtons)

This means with a coefficient of 1, you are applying the entire force of the car+suspension opposite of the slip direction. In real world applications, values higher than 1 are rare, but not impossible. For a tire on dry asphalt, values between .7 and .9 are realistic, so the default of 1.0 is preferable.

This value should not realistically not exceed 2.5, as strange behavior will begin to occur. E.g. you start to turn right, but because this value is so high, a large force is applied opposite of your

direction, and you begin to slide into the turn instead of away.

If you have maxed both values, you should then begin to raise the asymptote slip and value. Asymptote Slip should be between .5 and 2 m/s, and defines the coefficient of friction for any slip value past the Asymptote slip. If you find your vehicles behaves well until it break traction, at which point it acts like it is on ice, you should raise the Asymptote value. If you find that your vehicle is unable to drift, you should lower the value.

Forward Friction

Forward friction is identical to sideways friction, with the exception that this defines how much traction the wheel has in the direction of motion. If the values are too low, you vehicles will do burnouts and just spin the tires before moving forward, slowly. If it is too high, your vehicle may have a tendency to try and do a wheely, or worse, flip.

Additional Notes

Do not expect to be able to create a GTA clone, or other racing clone by simply adjusting these values. In most driving games, these values are constantly being changed in script for different speeds, terrains, and turning values. Additionally, if you are just applying a constant torque to the wheel colliders when a key is being pressed, your game will not behave realistically. In the real world, cars have torque curves and transmissions to change the torque applied to the wheels.

For best results, you should tune these values until you get a car the responds reasonably well, and then make changes to wheel torque, max turning angle, and friction values in script.

More information about wheel colliders can be found in Nvidia's documenation:
http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Vehicles.html

Trigger Colliders

Methods

- `OnTriggerEnter()`
- `OnTriggerStay()`
- `OnTriggerExit()`

You can make a Collider into a **Trigger** in order to use the `OnTriggerEnter()`, `OnTriggerStay()` and `OnTriggerExit()` methods. A Trigger Collider will not physically react to collisions, other GameObjects simply pass through it. They are useful for detecting when another GameObject is in a certain area or not, for example, when collecting an item, we may want to be able to just run through it but detect when this happens.

Trigger Collider Scripting

Example

The method below is an example of a trigger listener that detects when another collider enters the collider of a GameObject (such as a player). Trigger methods can be added to any script that is assigned to a GameObject.

```
void OnTriggerEnter(Collider other)
{
    //Check collider for specific properties (Such as tag=item or has component=item)
}
```

Read Collision online: https://riptutorial.com/unity3d/topic/4405/collision

Chapter 8: Communication with server

Examples

Get

Get is getting data from web server. and `new WWW("https://urlexample.com");` with a url but without a second parameter is doing a **Get**.

i.e.

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    public string url = "http://google.com";

    IEnumerator Start()
    {
        WWW www = new WWW(url); // One get.
        yield return www;
        Debug.Log(www.text); // The data of the url.
    }
}
```

Simple Post (Post Fields)

Every instance of **WWW** with a second parameter is a *post*.

Here is an example to post *user id* and *password* to server.

```
void Login(string id, string pwd)
{
    WWWForm dataParameters = new WWWForm();    // Create a new form.
    dataParameters.AddField("username", id);
    dataParameters.AddField("password", pwd);   // Add fields.
    WWW www = new WWW(url+"/account/login",dataParameters);
    StartCoroutine("PostdataEnumerator", www);
}

IEnumerator PostdataEnumerator(WWW www)
{
    yield return www;
    if (!string.IsNullOrEmpty(www.error))
    {
        Debug.Log(www.error);
    }
    else
    {
        Debug.Log("Data Submitted");
    }
}
```

Post (Upload A File)

Upload a file to server is also a post. You can easily upload a file through **WWW**, like the below:

Upload A Zip File To Server

```
string mainUrl = "http://server/upload/";
string saveLocation;

void Start()
{
    saveLocation = "ftp:///home/xxx/x.zip"; // The file path.
    StartCoroutine(PrepareFile());
}

// Prepare The File.
IEnumerator PrepareFile()
{
    Debug.Log("saveLoacation = " + saveLocation);

    // Read the zip file.
    WWW loadTheZip = new WWW(saveLocation);

    yield return loadTheZip;

    PrepareStepTwo(loadTheZip);
}

void PrepareStepTwo(WWW post)
{
    StartCoroutine(UploadTheZip(post));
}

// Upload.
IEnumerator UploadTheZip(WWW post)
{
    // Create a form.
    WWWForm form = new WWWForm();

    // Add the file.
    form.AddBinaryData("myTestFile.zip",post.bytes,"myFile.zip","application/zip");

    // Send POST request.
    string url = mainUrl;
    WWW POSTZIP = new WWW(url,form);

    Debug.Log("Sending zip...");
    yield return POSTZIP;
    Debug.Log("Zip sent!");
}
```

In this example, it use the **coroutine** to prepare and upload the file, if you want to know more about Unity coroutines, please visit Coroutines.

Sending a request to the server

There are many ways to communicate with servers using Unity as the client (some methodologies are better than others depending on your purpose). First, one must determine the need of the server to be able to effectively send operations to and from the server. For this example, we will send a few pieces of data to our server to be validated.

Most likely, the programmer will have setup some sort of handler on their server to receive events and respond back to the client accordingly - however that is out of the scope of this example.

C#:

```
using System.Net;
using System.Text;

public class TestCommunicationWithServer
{
    public string SendDataToServer(string url, string username, string password)
    {
        WebClient client = new WebClient();

        // This specialized key-value pair will store the form data we're sending to the
server
        var loginData = new System.Collections.Specialized.NameValueCollection();
        loginData.Add("Username", username);
        loginData.Add("Password", password);

        // Upload client data and receive a response
        byte[] opBytes = client.UploadValues(ServerIpAddress, "POST", loginData);

        // Encode the response bytes into a proper string
        string opResponse = Encoding.UTF8.GetString(opBytes);

        return opResponse;
    }
```

First thing one must do is toss in their using statements which allow us to use the WebClient and NameValueCollection classes.

For this example the SendDataToServer function takes in 3 (optional) string parameters:

1. Url of the server we're communicating with
2. First piece of data
3. Second piece of data we're sending to the server

The username and password is the optional data I am sending to the server. For this example we're using it to be then further validated from a database or any other external storage.

Now that we've setup our structure, we will instantiate a new WebClient to be used to actually send our data. Now we need to load our data into our NameValueCollection and upload the data to the server.

The UploadValues function takes in 3 necessary parameters as well:

1. IP address of server
2. HTTP method

3. Data you're sending (the username and password in our case)

This function returns a byte array of the response from the server. We need to encode the returned byte array it into a proper string to actual be able to manipulate and dissect the response.

One could do something like this:

```
if(opResponse.Equals(ReturnMessage.Success))
{
    Debug.Log("Unity client has successfully sent and validated data on server.");
}
```

Now you might still be confused so I guess I will give a brief explanation of how to handle a response server sided.

For this example I will be using PHP to handle the response from the client. I'd recommend using PHP as your back-end scripting language because it's super versatile, easy to use and most of all fast. There definitely are other ways to handle a response on a server but in my opinion PHP is by far the simplest and easiest implementation into Unity.

PHP:

```
// Check to see if the unity client send the form data
if(!isset($_REQUEST['Username']) || !isset($_REQUEST['Password']))
{
    echo "Empty";
}
else
{
    // Unity sent us the data - its here so do whatever you want

    echo "Success";

}
```

So this is the most important part - the 'echo'. When our client uploads the data to server, the client saves the response (or resource) into that byte array. Once the client has the response, you know the data has been validated and you can move on in the client once that event has happened. You also need to think about what type of data you're sending (to an extent), and how to minimize the amount you're actually sending.

So this is only one way of sending/receiving data from Unity - there are some other ways that may be more effective for you depending on your project.

Read Communication with server online: https://riptutorial.com/unity3d/topic/5578/communication-with-server

Chapter 9: Coroutines

Syntax

- public Coroutine StartCoroutine(IEnumerator routine);
- public Coroutine StartCoroutine(string methodName, object value = null);
- public void StopCoroutine(string methodName);
- public void StopCoroutine(IEnumerator routine);
- public void StopAllCoroutines();

Remarks

Performance considerations

It's best to use coroutines in moderation as the flexibility comes with a performance cost.

- Coroutines in great numbers demands more from the CPU than standard Update methods.
- There is an issue in some versions of Unity where coroutines produce garbage each update cycle due to Unity boxing the `MoveNext` return value. This was last observed in 5.4.0b13. (Bug report)

Reduce garbage by caching YieldInstructions

A common trick to reduce the garbage generated in coroutines is to cache the `YieldInstruction`.

```
IEnumerator TickEverySecond()
{
    var wait = new WaitForSeconds(1f); // Cache
    while(true)
    {
        yield return wait; // Reuse
    }
}
```

Yielding `null` produces no extra garbage.

Examples

Coroutines

First it's essential to understand that, game engines (such as Unity) work on a "frame based" paradigm.

Code is executed during every frame.

That includes Unity's own code, and your code.

When thinking about frames, it's important to understand that there is ***absolutely*** no guarantee of when frames happen. They ***do not*** happen on a regular beat. The gaps between frames could be, for example, 0.02632 then 0.021167 then 0.029778, and so on. In the example they are all "about" 1/50th of a second, but they are all different. And at any time, you may get a frame that takes much longer, or shorter; and your code may be executed at any time at all within the frame.

Bearing that in mind, you may ask: how do you access these frames in your code, in Unity?

Quite simply, you use either the Update() call, or, you use a coroutine. (Indeed - they are exactly the same thing: they allow code to be run every frame.)

The purpose of a coroutine is that:

you can run some code, and then, "stop and wait" ***until some future frame***.

You can wait until **the next frame**, you can wait for **a number of frames**, or you can wait for some ***approximate*** time in seconds in the future.

For example, you can wait for "about one second", meaning it will wait for about one second, and then put your code in some frame roughly one second from now. (And indeed, within that frame, the code could be run at any time, whatsoever.) To repeat: it will not be exactly one second. Accurate timing is meaningless in a game engine.

Inside a coroutine:

To wait one frame:

```
// do something
yield return null;  // wait until next frame
// do something
```

To wait three frames:

```
// do something
yield return null;  // wait until three frames from now
yield return null;
yield return null;
// do something
```

To wait **approximately** half a second:

```
// do something
yield return new WaitForSeconds (0.5f); // wait for a frame in about .5 seconds
// do something
```

Do something every single frame:

```
while (true)
{
```

```
    // do something
    yield return null;  // wait until the next frame
}
```

That example is literally identical to simply putting something inside Unity's "Update" call: the code at "do something" is run every frame.

Example

Attach Ticker to a `GameObject`. While that game object is active, the tick will run. Note that the script carefully stops the coroutine, when the game object becomes inactive; this is usually an important aspect of correctly engineering coroutine usage.

```
using UnityEngine;
using System.Collections;

public class Ticker:MonoBehaviour {

    void OnEnable()
    {
        StartCoroutine(TickEverySecond());
    }

    void OnDisable()
    {
        StopAllCoroutines();
    }

    IEnumerator TickEverySecond()
    {
        var wait = new WaitForSeconds(1f); // REMEMBER: IT IS ONLY APPROXIMATE
        while(true)
        {
            Debug.Log("Tick");
            yield return wait;  // wait for a frame, about 1 second from now
        }
    }
}
```

Ending a coroutine

Often you design coroutines to naturally end when certain goals are met.

```
IEnumerator TickFiveSeconds()
{
    var wait = new WaitForSeconds(1f);
    int counter = 1;
    while(counter < 5)
    {
        Debug.Log("Tick");
        counter++;
        yield return wait;
    }
    Debug.Log("I am done ticking");
```

```
}
```

To stop a coroutine from "inside" the coroutine, you cannot simply "return" as you would to leave early from an ordinary function. Instead, you use `yield break`.

```
IEnumerator ShowExplosions()
{
    ... show basic explosions
    if(player.xp < 100) yield break;
    ... show fancy explosions
}
```

You can also force all coroutines launched by the script to halt before finishing.

```
void OnDisable()
{
    // Stops all running coroutines
    StopAllCoroutines();
}
```

The method to stop a *specific* coroutine from the caller varies depending on how you started it.

If you started a coroutine by string name:

```
StartCoroutine("YourAnimation");
```

then you can stop it by calling StopCoroutine with the same string name:

```
StopCoroutine("YourAnimation");
```

Alternatively, you can keep a reference to *either* the `IEnumerator` returned by the coroutine method, *or* the `Coroutine` object returned by `StartCoroutine`, and call `StopCoroutine` on either of those:

```
public class SomeComponent : MonoBehaviour
{
    Coroutine routine;

    void Start () {
        routine = StartCoroutine(YourAnimation());
    }

    void Update () {
        // later, in response to some input...
        StopCoroutine(routine);
    }

    IEnumerator YourAnimation () { /* ... */ }
}
```

MonoBehaviour methods that can be Coroutines

There are three MonoBehaviour methods that can be made coroutines.

1. Start()
2. OnBecameVisible()
3. OnLevelWasLoaded()

This can be used to create, for example, scripts that execute only when the object is visible to a camera.

```
using UnityEngine;
using System.Collections;

public class RotateObject : MonoBehaviour
{
    IEnumerator OnBecameVisible()
    {
        var tr = GetComponent<Transform>();
        while (true)
        {
            tr.Rotate(new Vector3(0, 180f * Time.deltaTime));
            yield return null;
        }
    }

    void OnBecameInvisible()
    {
        StopAllCoroutines();
    }
}
```

Chaining coroutines

Coroutines can yield inside themselves, and wait for **other coroutines**.

So, you can chain sequences - "one after the other".

This is very easy, and is a basic, core, technique in Unity.

It's absolutely natural in games that certain things have to happen "in order". Almost every "round" of a game starts with a certain series of events happening, over a space of time, in some order. Here's how you might start a car race game:

```
IEnumerator BeginRace()
{
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
}
```

So, when you call BeginRace ...

```
 StartCoroutine(BeginRace());
```

It will run your "prepare race" routine. (Perhaps, flashing some lights and running some crowd noise, resetting scores and so on.) When that is finished, it will run your "countdown" sequence,

where you would animate perhaps a countdown on the UI. When that is finished, it will run your race-starting code, where you would perhaps run sound effects, start some AI drivers, move the camera in a certain way, and so on.

For clarity, understand that the three calls

```
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
```

must themselves **be in** a coroutine. That is to say, they must be in a function of the type `IEnumerator`. So in our example that's `IEnumerator BeginRace`. So, from "normal" code, you launch that coroutine with the `StartCoroutine` call.

```
 StartCoroutine(BeginRace());
```

To further understand chaining, here's a function which chains coroutines. You pass in an array of coroutines. The function runs as many coroutines as you pass, in order, one after the other.

```
// run various routines, one after the other
IEnumerator OneAfterTheOther( params IEnumerator[] routines )
{
    foreach ( var item in routines )
    {
        while ( item.MoveNext() ) yield return item.Current;
    }

    yield break;
}
```

Here's how you would call that...let's say you have three functions. Recall they must all be `IEnumerator`:

```
IEnumerator PrepareRace()
{
    // codesay, crowd cheering and camera pan around the stadium
    yield break;
}

IEnumerator Countdown()
{
    // codesay, animate your countdown on UI
    yield break;
}

IEnumerator StartRace()
{
    // codesay, camera moves and light changes and launch the AIs
    yield break;
}
```

You'd call it like this

```
StartCoroutine( MultipleRoutines( PrepareRace(), Countdown(), StartRace() ) );
```

or perhaps like this

```
IEnumerator[] routines = new IEnumerator[] {
     PrepareRace(),
     Countdown(),
     StartRace() };
StartCoroutine( MultipleRoutines( routines ) );
```

To repeat, one of the most basic requirements in games is that certain things happen one after the other "in a sequence" over time. You achieve that in Unity very simply, with

```
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
```

Ways to yield

You can wait until the next frame.

```
yield return null; // wait until sometime in the next frame
```

You can have multiple of these calls in a row, to simply wait for as many frames as desired.

```
//wait for a few frames
yield return null;
yield return null;
```

Wait for ***approximately*** n seconds. It is extremely important to understand this is only **a very approximate time**.

```
yield return new WaitForSeconds(n);
```

It is absolutely not possible to use the "WaitForSeconds" call for any form of accurate timing.

Often you want to chain actions. So, do something, and when that is finished do something else, and when that is finished do something else. To achieve that, wait for another coroutine:

```
yield return StartCoroutine(coroutine);
```

Understand that you can only call that from within a coroutine. So:

```
StartCoroutine(Test());
```

That's how you start a coroutine from a "normal" piece of code.

Then, inside that running coroutine:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

That will print A, start the long process, and **immediately print B**. It will **not** wait for the long process to finish. On the other hand:

```
Debug.Log("A");
yield return StartCoroutine(LongProcess());
Debug.Log("B");
```

That will print A, start the long process, **wait until it is finished**, and then print B.

It's always worth remembering that coroutines have absolutely no connection, in any way, to threading. With this code:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

it is easy to think of it as being "like" starting the LongProcess on another thread in the background. But that is absolutely incorrect. It is just a coroutine. Game engines are frame based, and "coroutines" in Unity simply allow you to access the frames.

It is very easy to wait for a web request to complete.

```
void Start() {
    string url = "http://google.com";
    WWW www = new WWW(url);
    StartCoroutine(WaitForRequest(www));
}

IEnumerator WaitForRequest(WWW www) {
    yield return www;

    if (www.error == null) {
        //use www.data);
    }
    else {
        //use www.error);
    }
}
```

For completeness: In very rare cases you use fixed update in Unity; there is a `WaitForFixedUpdate()` call which normally would never be used. There is a specific call (`WaitForEndOfFrame()` in the current version of Unity) which is used in certain situations in relation to generating screen captures during development. (The exact mechanism changes slightly as Unity evolves, so google for the latest info if relevant.)

Read Coroutines online: https://riptutorial.com/unity3d/topic/3415/coroutines

Chapter 10: CullingGroup API

Remarks

Since using CullingGroups is not always very straightforward, it may be helpful to encapsulate the bulk of the logic behind a manager class.

Below is a blueprint how such a manager might operate.

```
using UnityEngine;
using System;
public interface ICullingGroupManager
{
    int ReserveSphere();
    void ReleaseSphere(int sphereIndex);
    void SetPosition(int sphereIndex, Vector3 position);
    void SetRadius(int sphereIndex, float radius);
    void SetCullingEvent(int sphereIndex, Action<CullingGroupEvent> sphere);
}
```

The gist of it is that you reserve a culling sphere from the manager which returns the index of the reserved sphere. You then use the given index to manipulate your reserved sphere.

Examples

Culling object distances

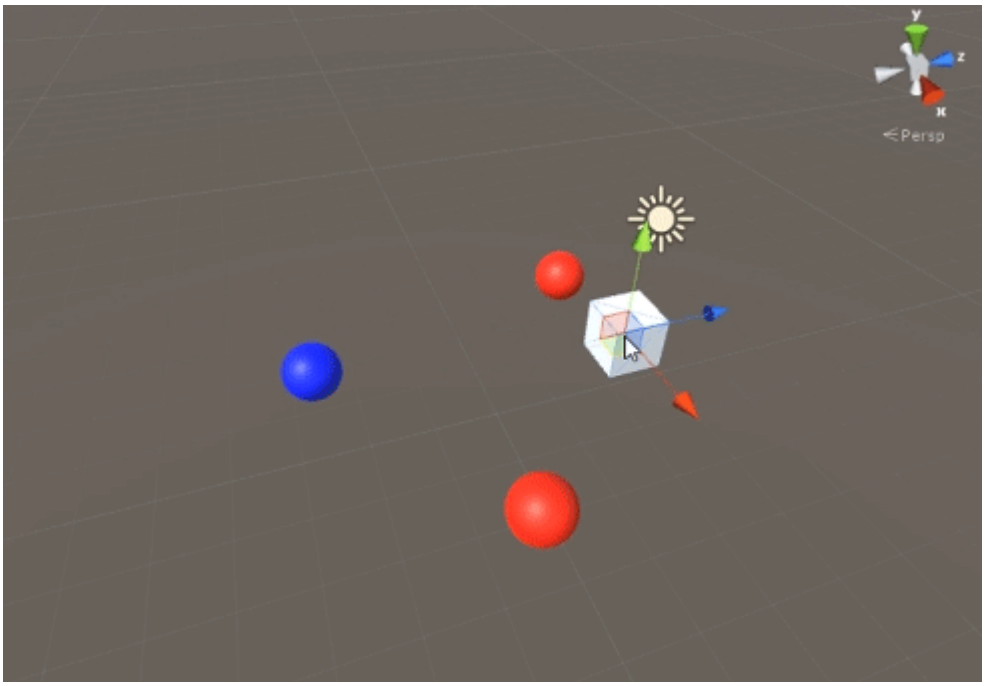
The following example illustrates how to use CullingGroups to get notifications according to the distance reference point.

> This script has been simplified for brevity and uses several performance heavy methods.

```
using UnityEngine;
using System.Linq;

public class CullingGroupBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;
    Transform[] meshTransforms;
    BoundingSphere[] cullingPoints;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();

        meshRenderers = FindObjectsOfType<MeshRenderer>()
                .Where((MeshRenderer m) => m.gameObject != this.gameObject)
                .ToArray();
```

```
        cullingPoints = new BoundingSphere[meshRenderers.Length];
        meshTransforms = new Transform[meshRenderers.Length];

        for (var i = 0; i < meshRenderers.Length; i++)
        {
            meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
            cullingPoints[i].position = meshTransforms[i].position;
            cullingPoints[i].radius = 4f;
        }

        localCullingGroup.onStateChanged = CullingEvent;
        localCullingGroup.SetBoundingSpheres(cullingPoints);
        localCullingGroup.SetBoundingDistances(new float[] { 0f, 5f });
        localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
        localCullingGroup.targetCamera = Camera.main;
    }

    void FixedUpdate()
    {
        localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
        for (var i = 0; i < meshTransforms.Length; i++)
        {
            cullingPoints[i].position = meshTransforms[i].position;
        }
    }

    void CullingEvent(CullingGroupEvent sphere)
    {
        Color newColor = Color.red;
        if (sphere.currentDistance == 1) newColor = Color.blue;
        if (sphere.currentDistance == 2) newColor = Color.white;
        meshRenderers[sphere.index].material.color = newColor;
    }

    void OnDisable()
    {
        localCullingGroup.Dispose();
    }
}
```

Add the script to a GameObject (in this case a cube) and hit Play. Every other GameObject in scene changes color according to their distance to the reference point.



Culling object visibility

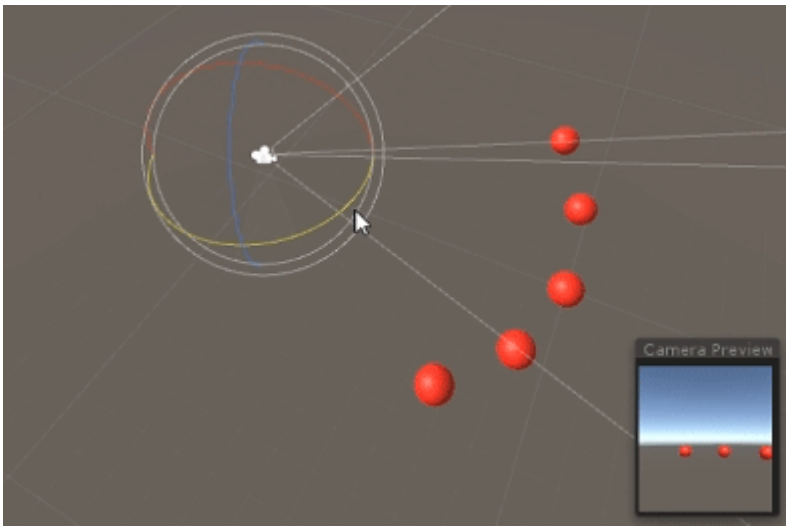
Following script illustrates how to receive events according to visibility to a set camera.

> This script uses several performance heavy methods for brevity.

```
using UnityEngine;
using System.Linq;

public class CullingGroupCameraBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();

        meshRenderers = FindObjectsOfType<MeshRenderer>()
            .Where((MeshRenderer m) => m.gameObject != this.gameObject)
            .ToArray();

        BoundingSphere[] cullingPoints = new BoundingSphere[meshRenderers.Length];
        Transform[] meshTransforms = new Transform[meshRenderers.Length];

        for (var i = 0; i < meshRenderers.Length; i++)
        {
            meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
            cullingPoints[i].position = meshTransforms[i].position;
            cullingPoints[i].radius = 4f;
        }

        localCullingGroup.onStateChanged = CullingEvent;
        localCullingGroup.SetBoundingSpheres(cullingPoints);
        localCullingGroup.targetCamera = Camera.main;
    }
```

```
    void CullingEvent(CullingGroupEvent sphere)
    {
        meshRenderers[sphere.index].material.color = sphere.isVisible ? Color.red :
Color.white;
    }

    void OnDisable()
    {
        localCullingGroup.Dispose();
    }
}
```

Add the script to scene and hit Play. All geometry in scene will change color based on their visibility.



> Similar effect can be achieved using the `MonoBehaviour.OnBecameVisible()` method if the object has a `MeshRenderer` component. Use CulingGroups when you need to cull empty GameObjects, `Vector3` coordinates, or when you want a centralised method of tracking object visibilities.

Bounding distances

You can add bounding distances on top of culling point radius. They are in a manner additional trigger conditions outside the culling points' main radius, like "close", "far" or "very far".

```
cullingGroup.SetBoundingDistances(new float[] { 0f, 10f, 100f});
```

> Bounding distances affect only when used with a distance reference point. They have no effect during camera culling.

Visualising bounding distances

What may initially cause confusion is how bounding distances are added on top of the sphere radiuses.

First, culling group calculates the *area* of both the bounding sphere and the bounding distance. The two areas are added together, and the result is the trigger area for the distance band. The radius of this area can be used to visualise the bounding distance field of effect.

```
float cullingPointArea = Mathf.PI * (cullingPointRadius * cullingPointRadius);
float boundingArea = Mathf.PI * (boundingDistance * boundingDistance);
float combinedRadius = Mathf.Sqrt((cullingPointArea + boundingArea) / Mathf.PI);
```

Read CullingGroup API online: https://riptutorial.com/unity3d/topic/4574/cullinggroup-api

Chapter 11: Design Patterns

Examples

Model View Controller (MVC) Design Pattern

The model view controller is a very common design pattern that has been around for quite some time. This pattern focuses on reducing *spaghetti* code by separating classes into functional parts. Recently I have been experimenting with this design pattern in Unity and would like to lay out a basic example.

A MVC design consists of three core parts: Model, View and Controller.

Model: The model is a class representing the data portion of your object. This could be a player, inventory or an entire level. If programmed correctly, you should be able to take this script and use it outside of Unity.

Note a few things about the Model:

- It should not inherit from Monobehaviour
- It should not contain Unity specific code for portability
- Since we are avoiding Unity API calls, this can hinder things like implicit converters in the Model class (workarounds are required)

Player.cs

```
using System;

public class Player
{
    public delegate void PositionEvent(Vector3 position);
    public event PositionEvent OnPositionChanged;

    public Vector3 position
    {
        get
        {
            return _position;
        }
        set
        {
            if (_position != value) {
                _position = value;
                if (OnPositionChanged != null) {
                    OnPositionChanged(value);
                }
            }
        }
    }
    private Vector3 _position;
}
```

Vector3.cs

A custom Vector3 class to use with our data model.

```
using System;

public class Vector3
{
    public float x;
    public float y;
    public float z;

    public Vector3(float x, float y, float z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

View: The view is a class representing the viewing portion tied to the model. This is an appropriate class to derive from Monobehaviour. This should contain code that interacts directly with Unity specific APIs including `OnCollisinEnter`, `Start`, `Update`, etc...

- Typically inherits from Monobehaviour
- Contains Unity specific code

PlayerView.cs

```
using UnityEngine;

public class PlayerView : Monobehaviour
{
    public void SetPosition(Vector3 position)
    {
        transform.position = position;
    }
}
```

Controller: The controller is a class that binds together both the Model and View. Controllers keep both Model and View in sync as well as drive interaction. The controller can listen for events from either partner and update accordingly.

- Binds both the Model and View by syncing state
- Can drive interaction between partners
- Controllers may or may not be portable (You might have to use Unity code here)
- If you decide to not make your controller portable, consider making it a Monobehaviour to help with editor inspecting

PlayerController.cs

```
using System;
```

```
public class PlayerController
{
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public PlayerController(Player model, PlayerView view)
    {
        this.model = model;
        this.view = view;

        this.model.OnPositionChanged += OnPositionChanged;
    }

    private void OnPositionChanged(Vector3 position)
    {
        // Sync
        Vector3 pos = this.model.position;

        // Unity call required here! (we lost portability)
        this.view.SetPosition(new UnityEngine.Vector3(pos.x, pos.y, pos.z));
    }

    // Calling this will fire the OnPositionChanged event
    private void SetPosition(Vector3 position)
    {
        this.model.position = position;
    }
}
```

Final Usage

Now that we have all of the main pieces, we can create a factory that will generate all three parts.

PlayerFactory.cs

```
using System;

public class PlayerFactory
{
    public PlayerController controller { get; private set; }
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public void Load()
    {
        // Put the Player prefab inside the 'Resources' folder
        // Make sure it has the 'PlayerView' Component attached
        GameObject prefab = Resources.Load<GameObject>("Player");
        GameObject instance = GameObject.Instantiate<GameObject>(prefab);
        this.model = new Player();
        this.view = instance.GetComponent<PlayerView>();
        this.controller = new PlayerController(model, view);
    }
}
```

And finally we can call the factory from a manager...

Manager.cs

```
using UnityEngine;

public class Manager : Monobehaviour
{
    [ContextMenu("Load Player")]
    private void LoadPlayer()
    {
        new PlayerFactory().Load();
    }
}
```

Attach the Manager script to an empty GameObject in the scene, right click the component and select "Load Player".

For more complex logic you can introduce inheritance with abstract base classes and interfaces for an improved architecture.

Read Design Patterns online: https://riptutorial.com/unity3d/topic/10842/design-patterns

Chapter 12: Extending the Editor

Syntax

- [MenuItem(string itemName)]
- [MenuItem(string itemName, bool isValidateFunction)]
- [MenuItem(string itemName, bool isValidateFunction, int priority)]
- [ContextMenu(string name)]
- [ContextMenuItem(string name, string function)]
- [DrawGizmo(GizmoType gizmo)]
- [DrawGizmo(GizmoType gizmo, Type drawnGizmoType)]

Parameters

Parameter	Details
MenuCommand	MenuCommand is used to extract the context for a MenuItem
MenuCommand.context	The object that is the target of the menu command
MenuCommand.userData	An int for passing custom information to a menu item

Examples

Custom Inspector

Using a custom inspector allows you to change the way a script is drawn in the Inspector. Sometimes you want to add extra information in the inspector for your script that isn't possible to do with a custom property drawer.

Below is a simple example of a custom object that with using a custom inspector can show more useful information.

```
using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif

public class InspectorExample : MonoBehaviour {

    public int Level;
    public float BaseDamage;

    public float DamageBonus {
        get {
            return Level / 100f * 50;
        }
```

```
    }

    public float ActualDamage {
        get {
            return BaseDamage + DamageBonus;
        }
    }
}

#if UNITY_EDITOR
[CustomEditor( typeof( InspectorExample ) )]
public class CustomInspector : Editor {

    public override void OnInspectorGUI() {
        base.OnInspectorGUI();

        var ie = (InspectorExample)target;

        EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );
        EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
    }
}
#endif
```

First we define our custom behaviour with some fields

```
public class InspectorExample : MonoBehaviour {
    public int Level;
    public float BaseDamage;
}
```

The fields shown above are automatically drawn (without custom inspector) when you are viewing the script in the Inspector window.

```
public float DamageBonus {
    get {
        return Level / 100f * 50;
    }
}

public float ActualDamage {
    get {
        return BaseDamage + DamageBonus;
    }
}
```

These properties are not automatically drawn by Unity. To show these properties in the Inspector view we have to use our Custom Inspector.

We first have to define our custom inspector like this

```
[CustomEditor( typeof( InspectorExample ) )]
public class CustomInspector : Editor {
```

The custom inspector has to derive from *Editor* and needs the *CustomEditor* attribute. The

parameter of the attribute is the type of the object this custom inspector should be used for.

Next up is the OnInspectorGUI method. This method gets called whenever the script is shown in the inspector window.

```
public override void OnInspectorGUI() {
    base.OnInspectorGUI();
}
```

We make a call to *base.OnInspectorGUI()* to let Unity handle the other fields that are in the script. If we would not call this we would have to do more work ourselves.

Next are our custom properties that we want to show

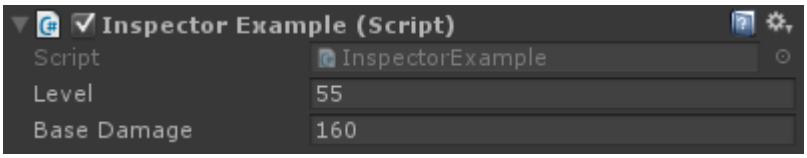
```
var ie = (InspectorExample)target;

EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );
EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
```

We have to create a temporary variable that holds target casted to our custom type (target is available because we derive from Editor).

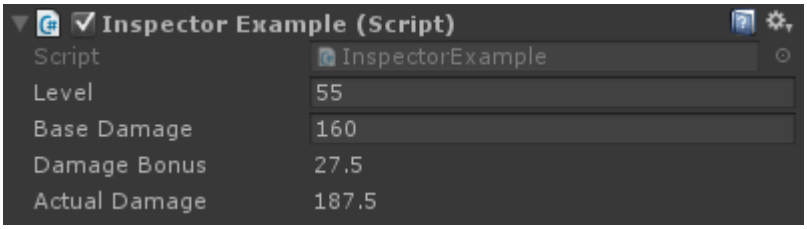
Next we can decide how to draw our properties, in this case two labelfields are enough since we just want to show the values and not be able to edit them.

Result

Before



After



Custom Property Drawer

Sometimes you have custom objects that contain data but do not derive from MonoBehaviour. Adding these objects as a field in a class that is MonoBehaviour will have no visual effect unless you write your own custom property drawer for the object's type.

Below is a simple example of a custom object, added to MonoBehaviour, and a custom property drawer for the custom object.

```
public enum Gender {
    Male,
    Female,
    Other
}

// Needs the Serializable attribute otherwise the CustomPropertyDrawer wont be used
[Serializable]
public class UserInfo {
    public string Name;
    public int Age;
    public Gender Gender;
}

// The class that you can attach to a GameObject
public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UInfo;
}

[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {

    public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
        // The 6 comes from extra spacing between the fields (2px each)
        return EditorGUIUtility.singleLineHeight * 4 + 6;
    }

    public override void OnGUI( Rect position, SerializedProperty property, GUIContent label )
{
        EditorGUI.BeginProperty( position, label, property );

        EditorGUI.LabelField( position, label );

        var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
        var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
        var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );

        EditorGUI.indentLevel++;

        EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
        EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
        EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

        EditorGUI.indentLevel--;

        EditorGUI.EndProperty();
    }
}
```

First off we define the custom object with all it's requirements. Just a simple class describing a user. This class is used in our PropertyDrawerExample class which we can add to a GameObject.

```
public enum Gender {
    Male,
    Female,
    Other
}

[Serializable]
public class UserInfo {
```

```
    public string Name;
    public int Age;
    public Gender Gender;
}

public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UInfo;
}
```

> The custom class needs the Serializable attribute otherwise the CustomPropertyDrawer will not be used

Next up is the CustomPropertyDrawer

First we have to define a class that derives from PropertyDrawer. The class definition also needs the CustomPropertyDrawer attribute. The parameter passed is the type of the object you want this drawer to be used for.

```
[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {
```

Next we override the GetPropertyHeight function. This allows us to define a custom height for our property. In this case we know that our property will have four parts: label, name, age, and gender. Therefore we use *EditorGUIUtility.singleLineHeight * 4*, we add another 6 pixels because we want to space each field with two pixels in between.

```
public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
    return EditorGUIUtility.singleLineHeight * 4 + 6;
}
```

Next is the actual OnGUI method. We start it off with *EditorGUI.BeginProperty([...])* and end the function with *EditorGUI.EndProperty()*. We do this so that if this property would be part of a prefab, the actual prefab overriding logic would work for everything in between those two methods.

```
public override void OnGUI( Rect position, SerializedProperty property, GUIContent label ) {
    EditorGUI.BeginProperty( position, label, property );
```

After that we show a label containing the name of the field and we already define the rectangles for our fields.

```
EditorGUI.LabelField( position, label );

var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );
```

> Every field is spaced by 16 + 2 pixels and the height is 16 (which is the same as EditorGUIUtility.singleLineHeight)

Next we indent the UI with one tab for a bit nicer layout, display the properties, un-indent the GUI, and end with *EditorGUI.EndProperty*.

```
EditorGUI.indentLevel++;

EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

EditorGUI.indentLevel--;

EditorGUI.EndProperty();
```

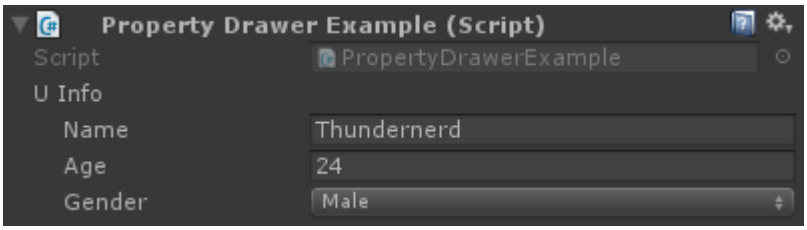
We display the fields by using *EditorGUI.PropertyField* which requires a rectangle for the position and a SerializedProperty for the property to show. We acquire the property by calling *FindPropertyRelative("...")* on the property passed in the *OnGUI* function. Note that these are case-sensitive and non-public properties cannot be found!

> For this example I am not saving the properties return from property.FindPropertyRelative("..."). You should save these in private fields in the class to prevent unnecessary calls

Result

Before

After



Menu Items

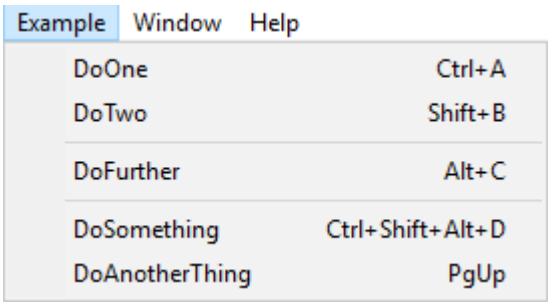
Menu items are a great way of adding custom actions to the editor. You can add menu items to the menu bar, have them as context-clicks on specific components, or even as context-clicks on fields in your scripts.

Below is an example of how you can apply menu items.

```
public class MenuItemsExample : MonoBehaviour {

    [MenuItem( "Example/DoSomething %#&d" )]
    private static void DoSomething() {
        // Execute some code
    }

    [MenuItem( "Example/DoAnotherThing", true )]
    private static bool DoAnotherThingValidator() {
        return Selection.gameObjects.Length > 0;
```

```
    }

    [MenuItem( "Example/DoAnotherThing _PGUP", false )]
    private static void DoAnotherThing() {
        // Execute some code
    }

    [MenuItem( "Example/DoOne %a", false, 1 )]
    private static void DoOne() {
        // Execute some code
    }

    [MenuItem( "Example/DoTwo #b", false, 2 )]
    private static void DoTwo() {
        // Execute some code
    }

    [MenuItem( "Example/DoFurther &c", false, 13 )]
    private static void DoFurther() {
        // Execute some code
    }

    [MenuItem( "CONTEXT/Camera/DoCameraThing" )]
    private static void DoCameraThing( MenuCommand cmd ) {
        // Execute some code
    }

    [ContextMenu( "ContextSomething" )]
    private void ContentSomething() {
        // Execute some code
    }

    [ContextMenuItem( "Reset", "ResetDate" )]
    [ContextMenuItem( "Set to Now", "SetDateToNow" )]
    public string Date = "";

    public void ResetDate() {
        Date = "";
    }

    public void SetDateToNow() {
        Date = DateTime.Now.ToString();
    }
}
```

Which looks like this



Let's go over the basic menu item. As you can see below you need to define a static function with a *MenuItem* attribute, which you pass a string as the title for the menu item. You can put your

menu item multiple levels deep by adding a / into the name.

```
[MenuItem( "Example/DoSomething %#&d" )]
private static void DoSomething() {
    // Execute some code
}
```

> You cannot have a menu item at top-level. Your menu items need to be in a submenu!

The special characters at the end of the MenuItem's name are for shortcut keys, these are not a requirement.

There are special characters that you can use for your shortcut keys, these are:

- % - Ctrl on Windows, Cmd on OS X
- # - Shift
- & - Alt

That means that the shortcut *%#&d* stands for ctrl+shift+alt+D on Windows, and cmd+shift+alt+D on OS X.

If you wish to use a shortcut without any special keys, so for instance just the 'D' key, you can prepend the _ (underscore) character to the shortcut key that you wish to use.

There are some other special keys that are supported, which are:

- LEFT, RIGHT, UP, DOWN - for the arrow keys
- F1..F12 - for the function keys
- HOME, END, PGUP, PGDN - for the navigation keys

> Shortcut keys need to be separated from any other text with a space

Next are validator menu items. Validator menu items allow menu items to be disabled (grayed-out, non-clickable) when the condition is not met. An example for this could be that your menu item acts on the current selection of GameObjects, which you can check for in the validator menu item.

```
[MenuItem( "Example/DoAnotherThing", true )]
private static bool DoAnotherThingValidator() {
    return Selection.gameObjects.Length > 0;
}

[MenuItem( "Example/DoAnotherThing _PGUP", false )]
private static void DoAnotherThing() {
    // Execute some code
}
```

For a validator menu item to work you need to create two static functions, both with the MenuItem attribute and the same name (shortcut key doesn't matter). The difference between them is that you're marking them as a validator function or not by passing a boolean parameter.

You can also define the order of the menu items by adding a priority. The priority is defined by an

integer that you pass as the third parameter. The smaller the number the higher up in the list, the bigger the number the lower in the list. You can add a separator in between two menu items by making sure there is at least 10 digits in between the priority of the menu items.

```
[MenuItem( "Example/DoOne %a", false, 1 )]
private static void DoOne() {
    // Execute some code
}

[MenuItem( "Example/DoTwo #b", false, 2 )]
private static void DoTwo() {
    // Execute some code
}

[MenuItem( "Example/DoFurther &c", false, 13 )]
private static void DoFurther() {
    // Execute some code
}
```

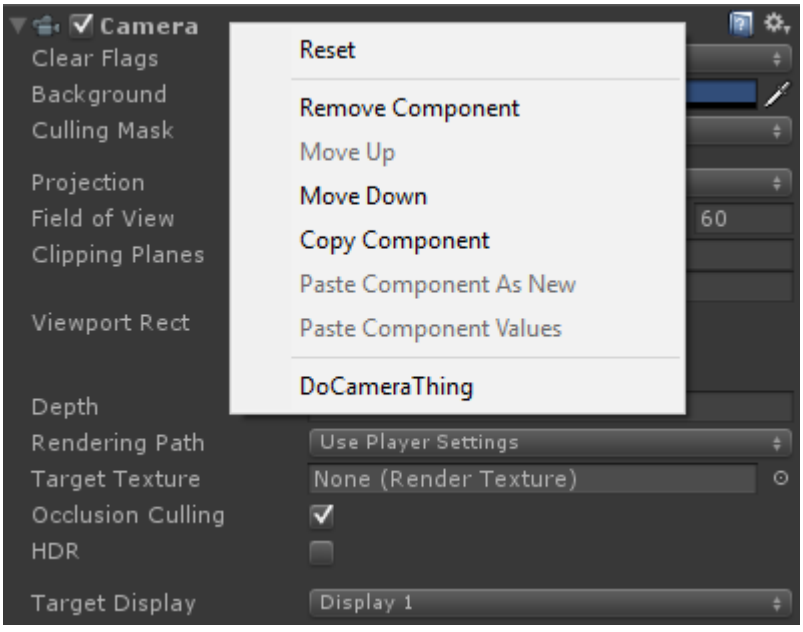
> If you have a menu list that has a combination of prioritized and non-prioritized items, the non-prioritized will be separated from the prioritized items.

Next is adding a menu item to the context menu of an already existing component. You have to start the name of the MenuItem with CONTEXT (case sensitive), and have your function take in a MenuCommand parameter.

The following snippet will add a context menu item to the Camera component.

```
[MenuItem( "CONTEXT/Camera/DoCameraThing" )]
private static void DoCameraThing( MenuCommand cmd ) {
    // Execute some code
}
```

Which looks like this



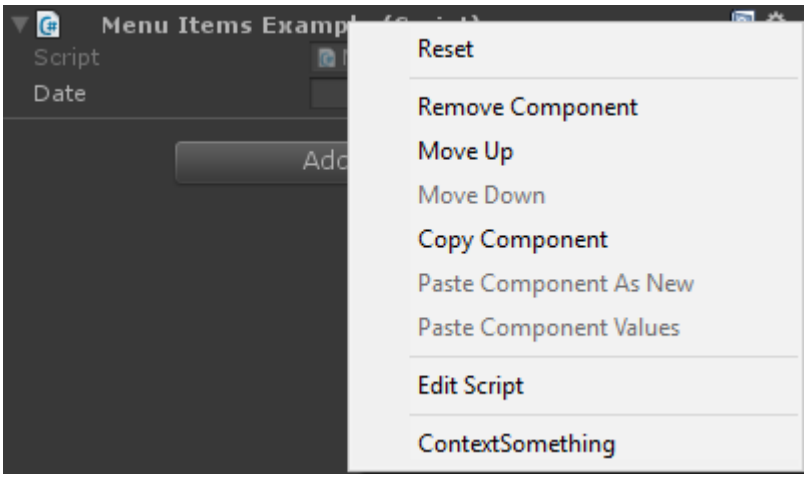
The MenuCommand parameter gives you access to the component value and any userdata that

gets send with it.

You can also add a context menu item to your own components by using the ContextMenu attribute. This attribute only takes a name, no validation or priority, and has to be part of a non-static method.

```
[ContextMenu( "ContextSomething" )]
private void ContentSomething() {
    // Execute some code
}
```

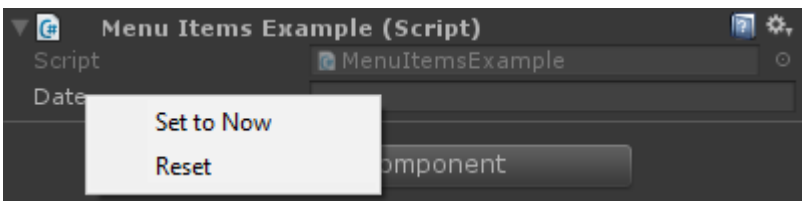
Which looks like this



You can also add context menu items to fields in your own component. These menu items will appear when you context-click on the field that they belong to and can execute methods that you have defined in that component. This way you can add for instance default values, or the current date, as shown below.

```
[ContextMenuItem( "Reset", "ResetDate" )]
[ContextMenuItem( "Set to Now", "SetDateToNow" )]
public string Date = "";

public void ResetDate() {
    Date = "";
}

public void SetDateToNow() {
    Date = DateTime.Now.ToString();
}
```

Which looks like this



Gizmos

Gizmos are used for drawing shapes in the scene view. You can use these shapes to draw extra information about your GameObjects, for instance the frustum they have or the detection range.

Below are two examples on how to do this

Example One

This example uses the *OnDrawGizmos* and *OnDrawGizmosSelected* (magic) methods.

```
public class GizmoExample : MonoBehaviour {

    public float GetDetectionRadius() {
        return 12.5f;
    }

    public float GetFOV() {
        return 25f;
    }

    public float GetMaxRange() {
        return 6.5f;
    }

    public float GetMinRange() {
        return 0;
    }

    public float GetAspect() {
        return 2.5f;
    }

    public void OnDrawGizmos() {
        var gizmoMatrix = Gizmos.matrix;
        var gizmoColor = Gizmos.color;

        Gizmos.matrix = Matrix4x4.TRS( transform.position, transform.rotation,
transform.lossyScale );
        Gizmos.color = Color.red;
        Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect()
);

        Gizmos.matrix = gizmoMatrix;
        Gizmos.color = gizmoColor;
    }

    public void OnDrawGizmosSelected() {
        Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
    }
}
```

In this example we have two methods for drawing gizmos, one that draws when the object is active (OnDrawGizmos) and one for when the object is selected in the hierarchy (OnDrawGizmosSelected).

```
public void OnDrawGizmos() {
```

```
    var gizmoMatrix = Gizmos.matrix;
    var gizmoColor = Gizmos.color;

    Gizmos.matrix = Matrix4x4.TRS( transform.position, transform.rotation,
transform.lossyScale );
    Gizmos.color = Color.red;
    Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect() );

    Gizmos.matrix = gizmoMatrix;
    Gizmos.color = gizmoColor;
}
```

First we save the gizmo matrix and color because we're going to change it and want to revert it back when we are done to not affect any other gizmo drawing.

Next we want to draw the frustum that our object has, however, we need to change the Gizmos' matrix so that it matches the position, rotation, and scale. We also set the Gizmos' color to red to emphasize the frustum. When this is done we can call *Gizmos.DrawFrustum* to draw the frustum in the scene view.

When we are done drawing what we want to draw, we reset the Gizmos' matrix and color.

```
public void OnDrawGizmosSelected() {
    Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
}
```

We also want to draw a detection range when we select our GameObject. This is done through the *Handles* class since the *Gizmos* class doesn't have any methods for discs.

Using this form of drawing gizmos results into the output shown below.

Example two

This example uses the *DrawGizmo* attribute.

```
public class GizmoDrawerExample {

    [DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]
    public static void DrawGizmo( GizmoExample obj, GizmoType type ) {
        var gizmoMatrix = Gizmos.matrix;
        var gizmoColor = Gizmos.color;

        Gizmos.matrix = Matrix4x4.TRS( obj.transform.position, obj.transform.rotation,
obj.transform.lossyScale );
        Gizmos.color = Color.red;
        Gizmos.DrawFrustum( Vector3.zero, obj.GetFOV(), obj.GetMaxRange(), obj.GetMinRange(),
obj.GetAspect() );

        Gizmos.matrix = gizmoMatrix;
        Gizmos.color = gizmoColor;

        if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {
            Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius()
```

```
);
            }
        }
}
```

This way allows you to separate the gizmo calls from your script. Most of this uses the same code as the other example except for two things.

```
[DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]
public static void DrawGizmo( GizmoExample obj, GizmoType type ) {
```

You need to use the DrawGizmo attribute which takes the enum GizmoType as the first parameter and a Type as the second parameter. The Type should be the type you want to use for drawing the gizmo.

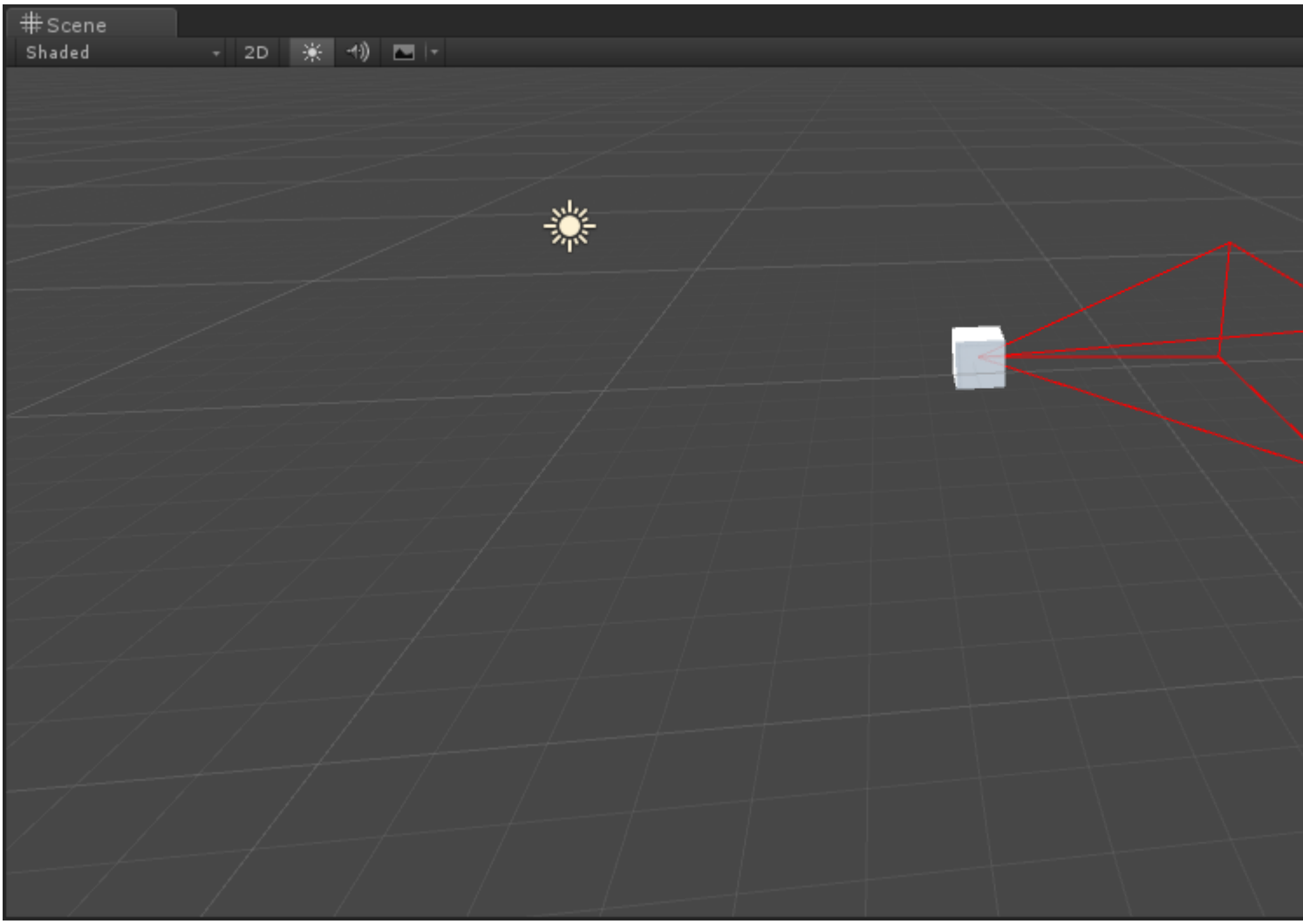
The method for drawing the gizmo needs to be static, public or non-public, and can be named whatever you want. The first parameter is the type, which should match the type passed as the second parameter in the attribute, and the second parameter is the enum GizmoType which describes the current state of your object.

```
if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {
    Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius() );
}
```

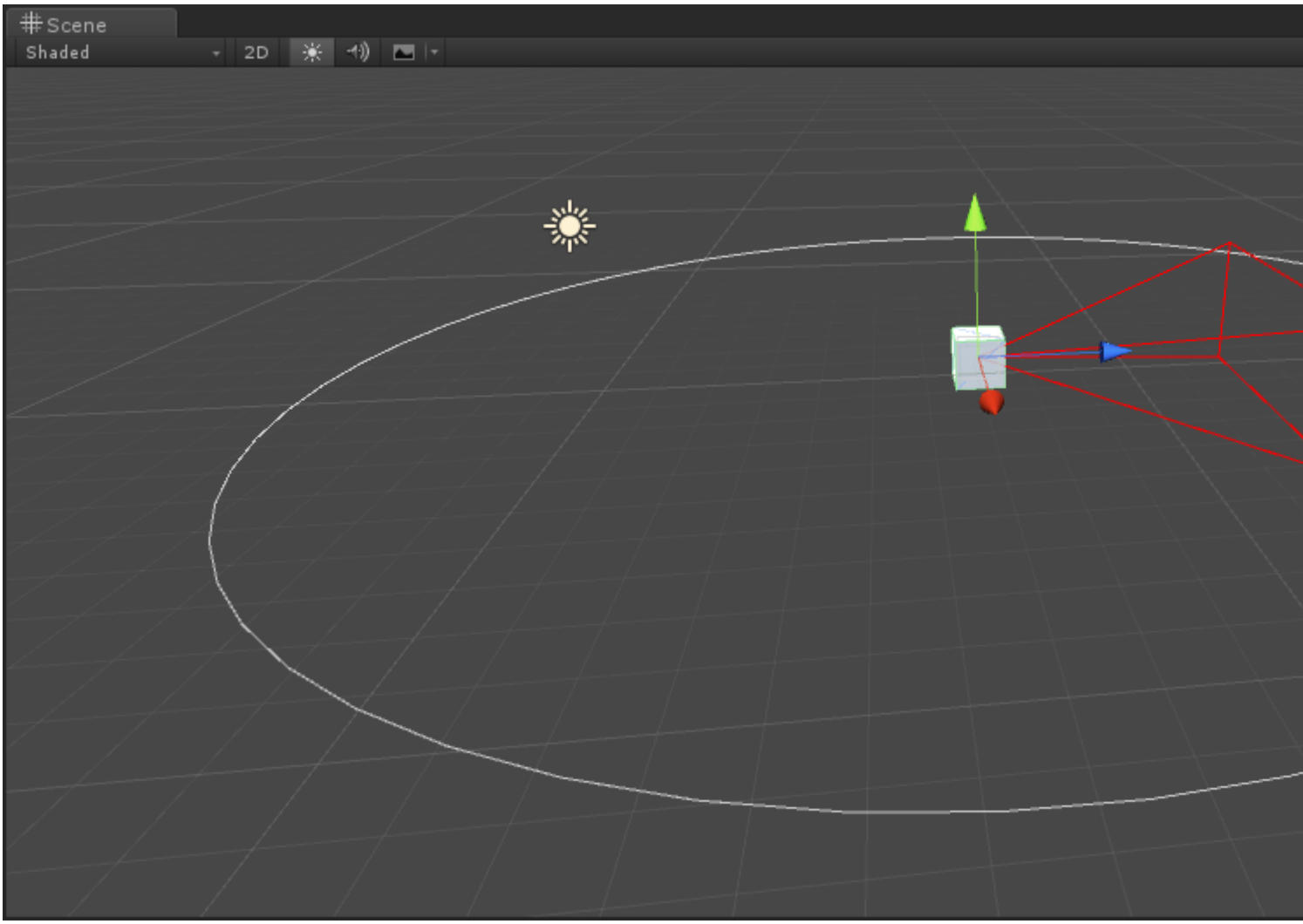
The other difference is that for checking what the GizmoType of the object is, you need to do an AND check on the parameter and the type you want.

Result

Not selected



Selected



Editor Window

Why an Editor Window?

As you may have seen, you can do a lot of things in a custom inspector (if you don't know what a custom inspector is, check the example here : http://www.riptutorial.com/unity3d/topic/2506/extending-the-editor. But at one point you may want to implement a configuration panel, or a customized asset palette. In those cases you are going to use an EditorWindow. Unity UI itself is composed of Editor Windows ; you can open them (usually through the top bar), tab them, etc.

Create a basic EditorWindow

Simple Example

Creating an custom editor window is fairly simple. All you need to do is extend the EditorWindow class and use the Init() and OnGUI() methods. Here is a simple example :

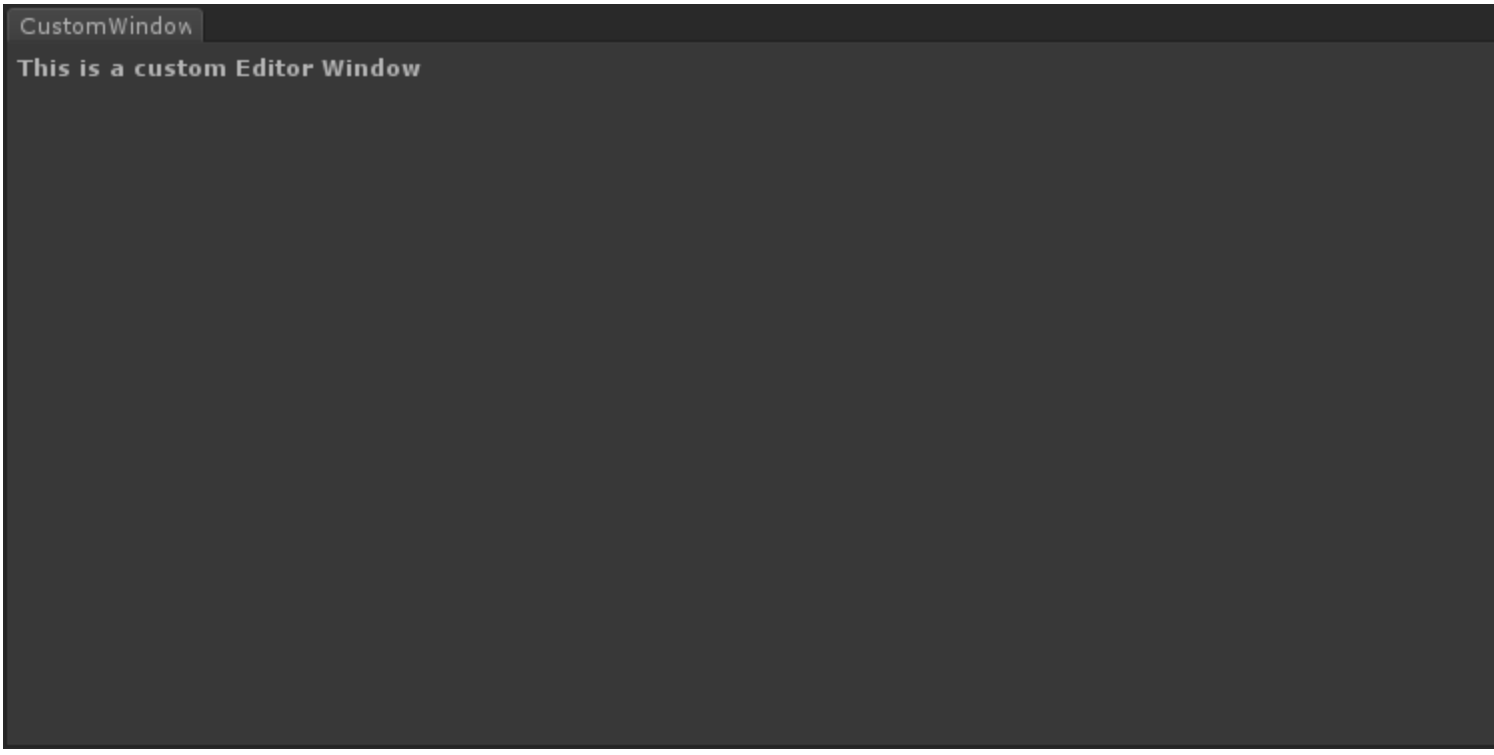
```
using UnityEngine;
using UnityEditor;
```

```
public class CustomWindow : EditorWindow
{
    // Add menu named "Custom Window" to the Window menu
    [MenuItem("Window/Custom Window")]
    static void Init()
    {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
        window.Show();
    }

    void OnGUI()
    {
        GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);
    }
}
```

The 3 important points are :

1. Don't forget to extend EditorWindow
2. Use the Init() as provided in the example. EditorWindow.GetWindow is checking if a CustomWindow is already created. If not, it will create a new instance. Using this you ensure that you don't have several instances of your window at the same time
3. Use OnGUI() like usual to display information in your window

The final result will look like this :



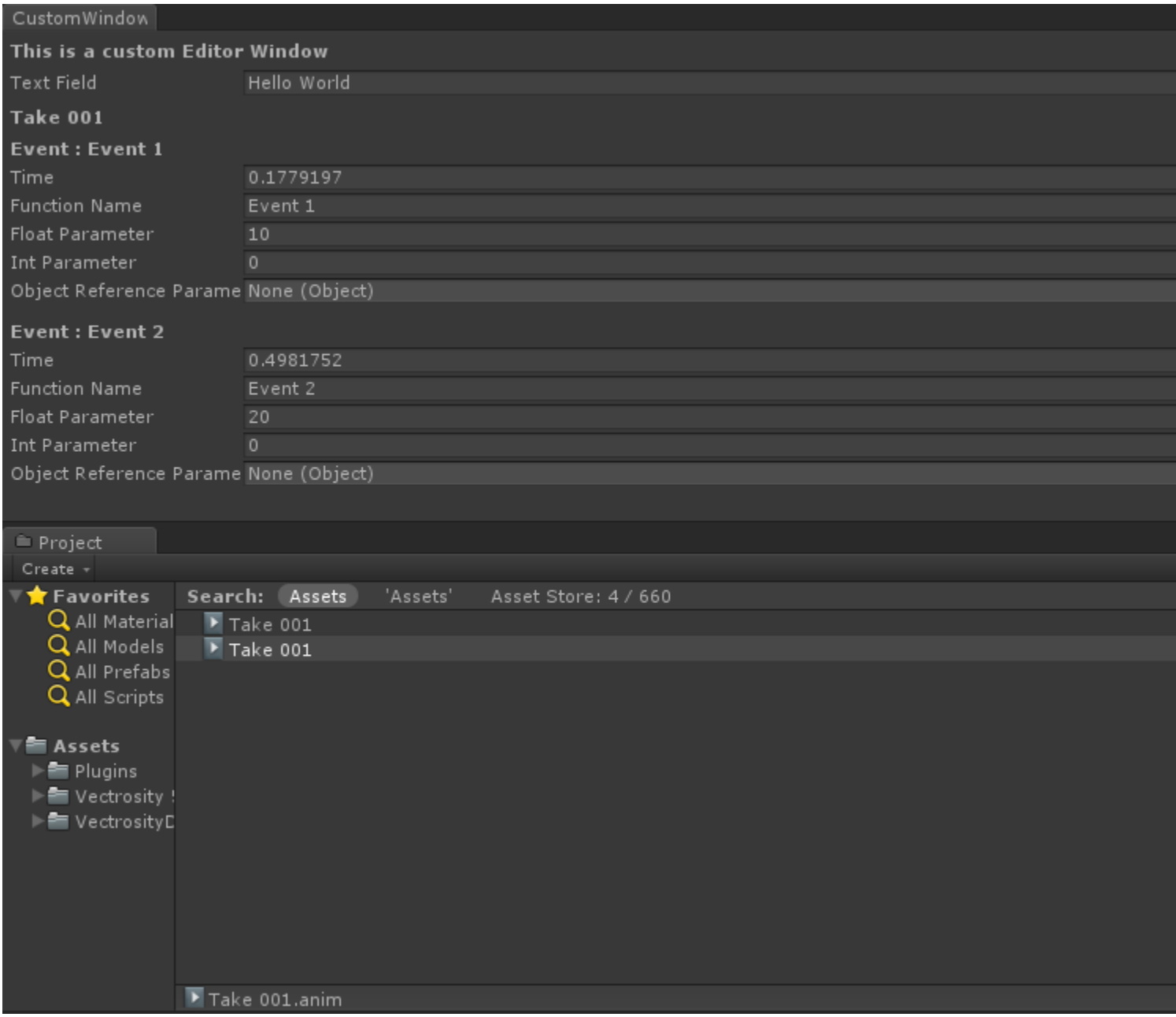
Going deeper

Of course you will probably want to manage or modify some assets using this EditorWindow. Here is an example using the Selection class (to get the active Selection) and modifying the selected asset properties via SerializedObject and SerializedProperty.

```
    using System.Linq;
    using UnityEngine;
    using UnityEditor;

    public class CustomWindow : EditorWindow
    {
        private AnimationClip _animationClip;
        private SerializedObject _serializedClip;
        private SerializedProperty _events;

        private string _text = "Hello World";

        // Add menu named "Custom Window" to the Window menu
        [MenuItem("Window/Custom Window")]
        static void Init()
        {
            // Get existing open window or if none, make a new one:
            CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
            window.Show();
        }

        void OnGUI()
        {
            GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);

            // You can use EditorGUI, EditorGUILayout and GUILayout classes to display
anything you want
            // A TextField example
            _text = EditorGUILayout.TextField("Text Field", _text);

            // Note that you can modify an asset or a gameobject using an EditorWindow. Here
is a quick example with an AnimationClip asset
            // The _animationClip, _serializedClip and _events are set in OnSelectionChange()

            if (_animationClip == null || _serializedClip == null || _events == null) return;

            // We can modify our serializedClip like we would do in a Custom Inspector. For
example we can grab its events and display their information

            GUILayout.Label(_animationClip.name, EditorStyles.boldLabel);

            for (var i = 0; i < _events.arraySize; i++)
            {
                EditorGUILayout.BeginVertical();

                EditorGUILayout.LabelField(
                    "Event : " +
_events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName").stringValue,
                    EditorStyles.boldLabel);

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("time"),
true,
                    GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName"),

                    true, GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("floatParameter"),

                    true, GUILayout.ExpandWidth(true));
```

```
                EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRe
lative("intParameter"),
                    true, GUILayout.ExpandWidth(true));
                EditorGUILayout.PropertyField(

_events.GetArrayElementAtIndex(i).FindPropertyRelative("objectReferenceParameter"), true,
                    GUILayout.ExpandWidth(true));

                EditorGUILayout.Separator();
                EditorGUILayout.EndVertical();
            }

            // Of course we need to Apply the modified properties. We don't our changes won't
be saved
            _serializedClip.ApplyModifiedProperties();
        }

        /// This Message is triggered when the user selection in the editor changes. That's
when we should tell our Window to Repaint() if the user selected another AnimationClip
        private void OnSelectionChange()
        {
            _animationClip =
                Selection.GetFiltered(typeof(AnimationClip),
SelectionMode.Assets).FirstOrDefault() as AnimationClip;
            if (_animationClip == null) return;

            _serializedClip = new SerializedObject(_animationClip);
            _events = _serializedClip.FindProperty("m_Events");
            Repaint();
        }
    }
```

Here is the result :



Advanced topics

You can do some really advanced things in the editor, and the EditorWindow class is perfect for displaying large amount of information. Most advanced assets on the Unity Asset Store (such as NodeCanvas or PlayMaker) use EditorWindow for displaying for custom views.

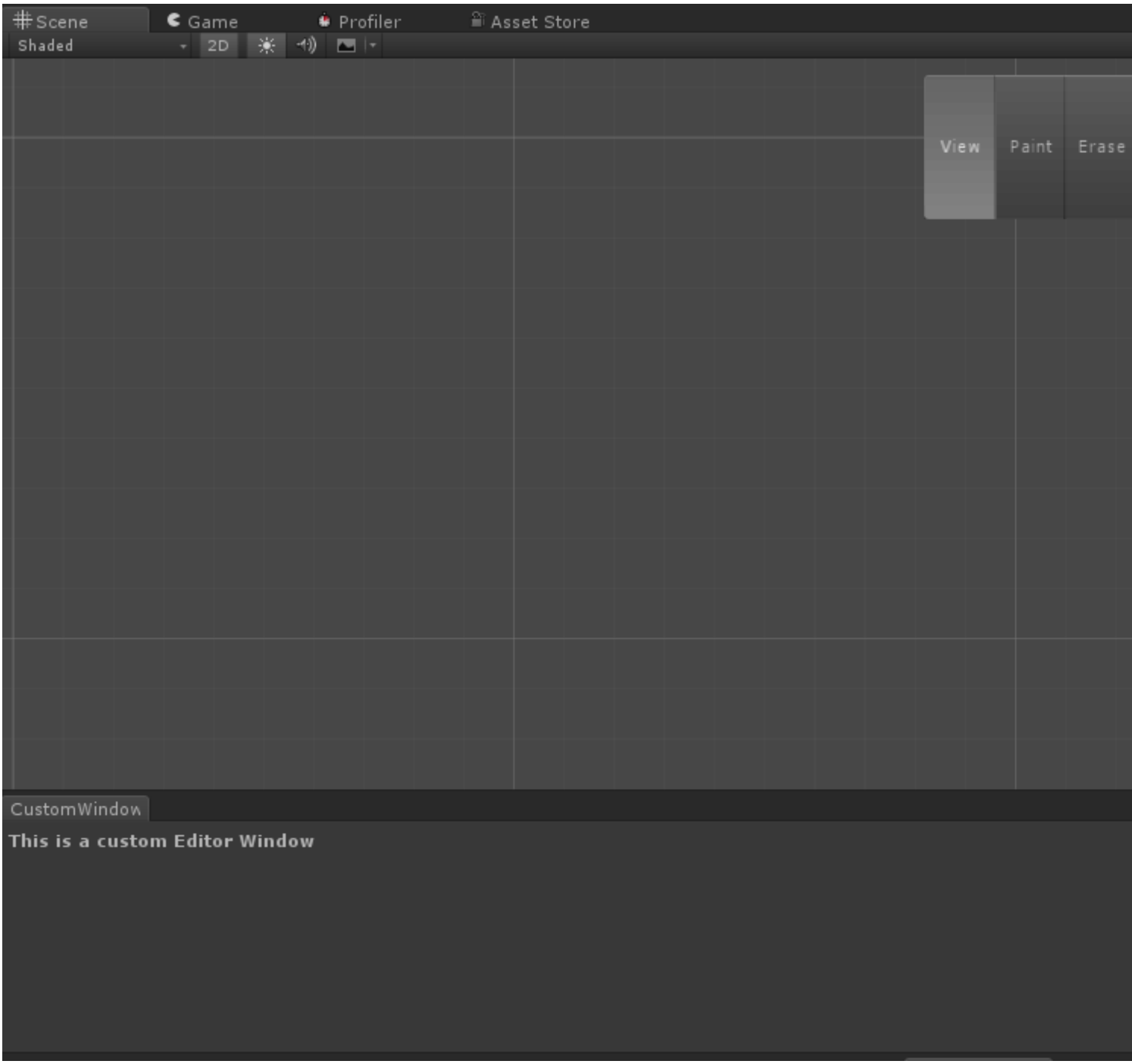
Drawing in the SceneView

One interesting thing to do with an EditorWindow is to display information directly in your SceneView. This way you can create a fully customized map/world editor, for example, using your custom EditorWindow as an asset palette and listening to clicks in the SceneView to instantiate new objects. Here is an example :

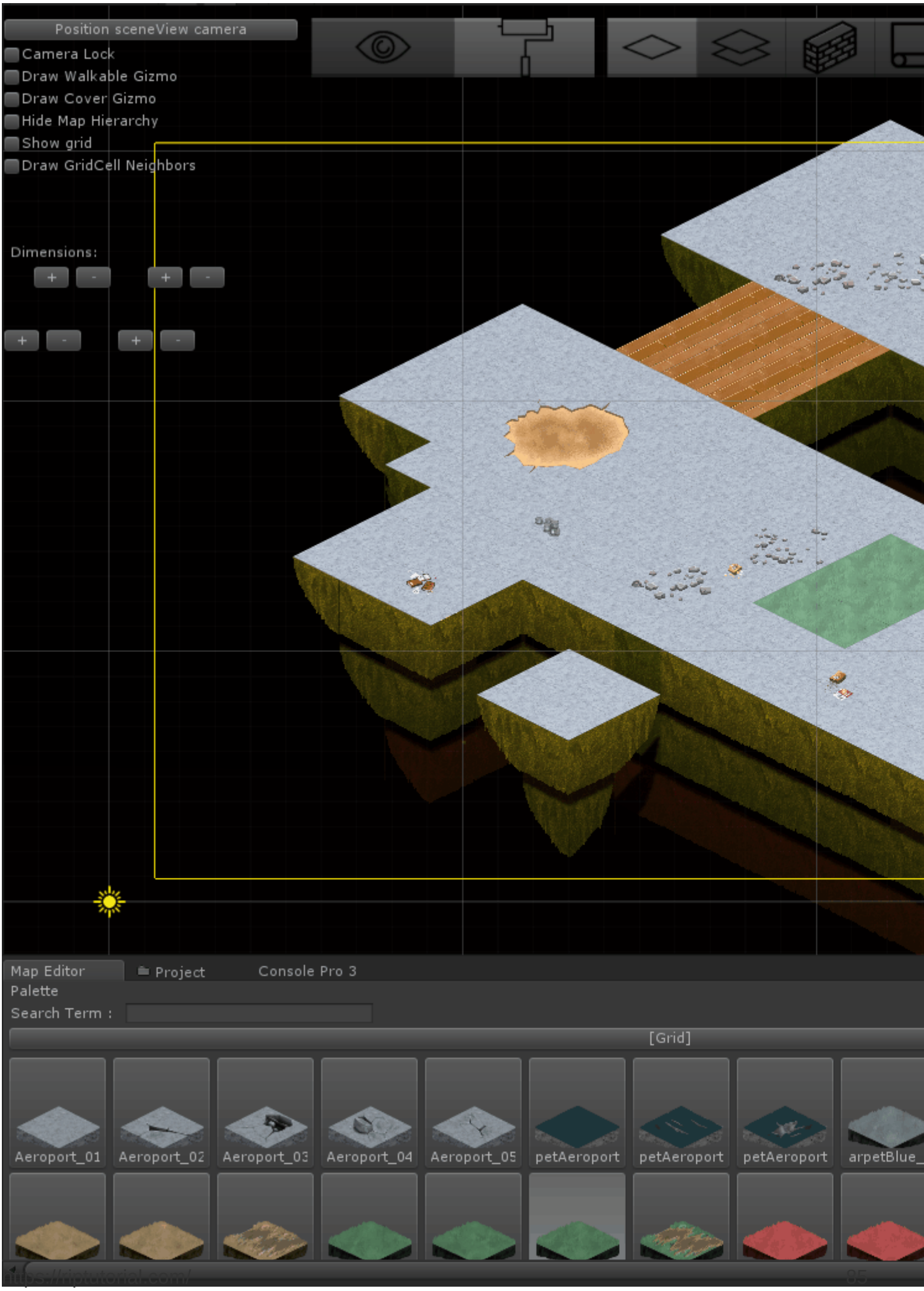
```
using UnityEngine;
using System;
```

```
using UnityEditor;

public class CustomWindow : EditorWindow {

    private enum Mode {
        View = 0,
        Paint = 1,
        Erase = 2
    }

    private Mode CurrentMode = Mode.View;

    [MenuItem ("Window/Custom Window")]
    static void Init () {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow)EditorWindow.GetWindow (typeof (CustomWindow));
        window.Show();
    }

    void OnGUI () {
        GUILayout.Label ("This is a custom Editor Window", EditorStyles.boldLabel);
    }

    void OnEnable() {
        SceneView.onSceneGUIDelegate = SceneViewGUI;
        if (SceneView.lastActiveSceneView) SceneView.lastActiveSceneView.Repaint();
    }

    void SceneViewGUI(SceneView sceneView) {
        Handles.BeginGUI();
        // We define the toolbars' rects here
        var ToolBarRect = new Rect((SceneView.lastActiveSceneView.camera.pixelRect.width / 6),
10, (SceneView.lastActiveSceneView.camera.pixelRect.width * 4 / 6) ,
SceneView.lastActiveSceneView.camera.pixelRect.height / 5);
        GUILayout.BeginArea(ToolBarRect);
        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
         CurrentMode = (Mode) GUILayout.Toolbar(
            (int) CurrentMode,
            Enum.GetNames(typeof(Mode)),
            GUILayout.Height(ToolBarRect.height));
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();
        GUILayout.EndArea();
        Handles.EndGUI();
    }
}
```

This will display the a toolbar directly in your SceneView



Here is a quick glimpse of how far you can go :


Position sceneView camera
Camera Lock
Draw Walkable Gizmo
Draw Cover Gizmo
Hide Map Hierarchy
Show grid
Draw GridCell Neighbors
Dimensions:
Map Editor
Project
Console Pro 3
Palette
Search Term :
[Grid]
Aeroport_01
Aeroport_02
Aeroport_03
Aeroport_04
Aeroport_05
petAeroport
petAeroport
petAeroport
arpetBlue_

Chapter 13: Finding and collecting GameObjects

Syntax

- public static GameObject Find(string name);
- public static GameObject FindGameObjectWithTag(string tag);
- public static GameObject[] FindGameObjectsWithTag(string tag);
- public static Object FindObjectOfType(Type type);
- public static Object[] FindObjectsOfType(Type type);

Remarks

Which method to use

Be careful while looking for GameObjects at runtime, as this can be resource consuming. Especially : don't run FindObjectOfType or Find in Update, FixedUpdate or more generally in a method called one or more time per frame.

- Call runtime methods `FindObjectOfType` and `Find` only when necessary
- `FindGameObjectWithTag` has very good performance compared to other string based methods. Unity keeps separate tabs on tagged objects and queries those instead of the entire scene.
- For "static" GameObjects (such as UI elements and prefabs) created in the editor use serializable GameObject reference in the editor
- Keep your lists of GameObjects in List or Arrays that you manage yourself
- In general, if you instantiate a lot of GameObjects of the same type take a look at Object Pooling
- Cache your search results to avoid running the expensive search methods again and again.

Going deeper

Besides the methods that come with Unity, it's relatively easy to design your own search and collection methods.

- In case of `FindObjectsOfType()`, you could have your scripts keep a list of themselves in a `static` collection. It is far faster to iterate a ready list of objects than to search and inspect objects from the scene.

- Or make a script that stores their instances in a string based `Dictionary`, and you have a simple tagging system you can expand upon.

Examples

Searching by GameObject's name

```
var go = GameObject.Find("NameOfTheObject");
```

Pros	Cons
Easy to use	Performance degrades along the number of gameobjects in scene
	Strings are *weak references* and suspect to user errors

Searching by GameObject's tags

```
var go = GameObject.FindGameObjectWithTag("Player");
```

Pros	Cons
Possible to search both single objects and entire groups	Strings are weak references and suspect to user errors.
Relatively fast and efficient	Code is not portable as tags are hard coded in scripts.

Inserted to scripts in Edit Mode

```
[SerializeField]
GameObject[] gameObjects;
```

Pros	Cons
Great performance	Object collection is static
Portable code	Can only refer to GameObjects from the same scene

Finding GameObjects by MonoBehaviour scripts

```
ExampleScript script = GameObject.FindObjectOfType<ExampleScript>();
GameObject go = script.gameObject;
```

`FindObjectOfType()` returns `null` if none is found.

Pros	Cons
Strongly typed	Performance degrades along the number of gameobjects needed to evaluate
Possible to search both single objects	

Pros	Cons
and entire groups	

Find GameObjects by name from child objects

```
Transform tr = GetComponent<Transform>().Find("NameOfTheObject");
GameObject go = tr.gameObject;
```

`Find` returns `null` if none is found

Pros	Cons
Limited, well defined search scope	Strings are weak references

Read Finding and collecting GameObjects online: https://riptutorial.com/unity3d/topic/3793/finding-and-collecting-gameobjects

Chapter 14: How to use asset packages

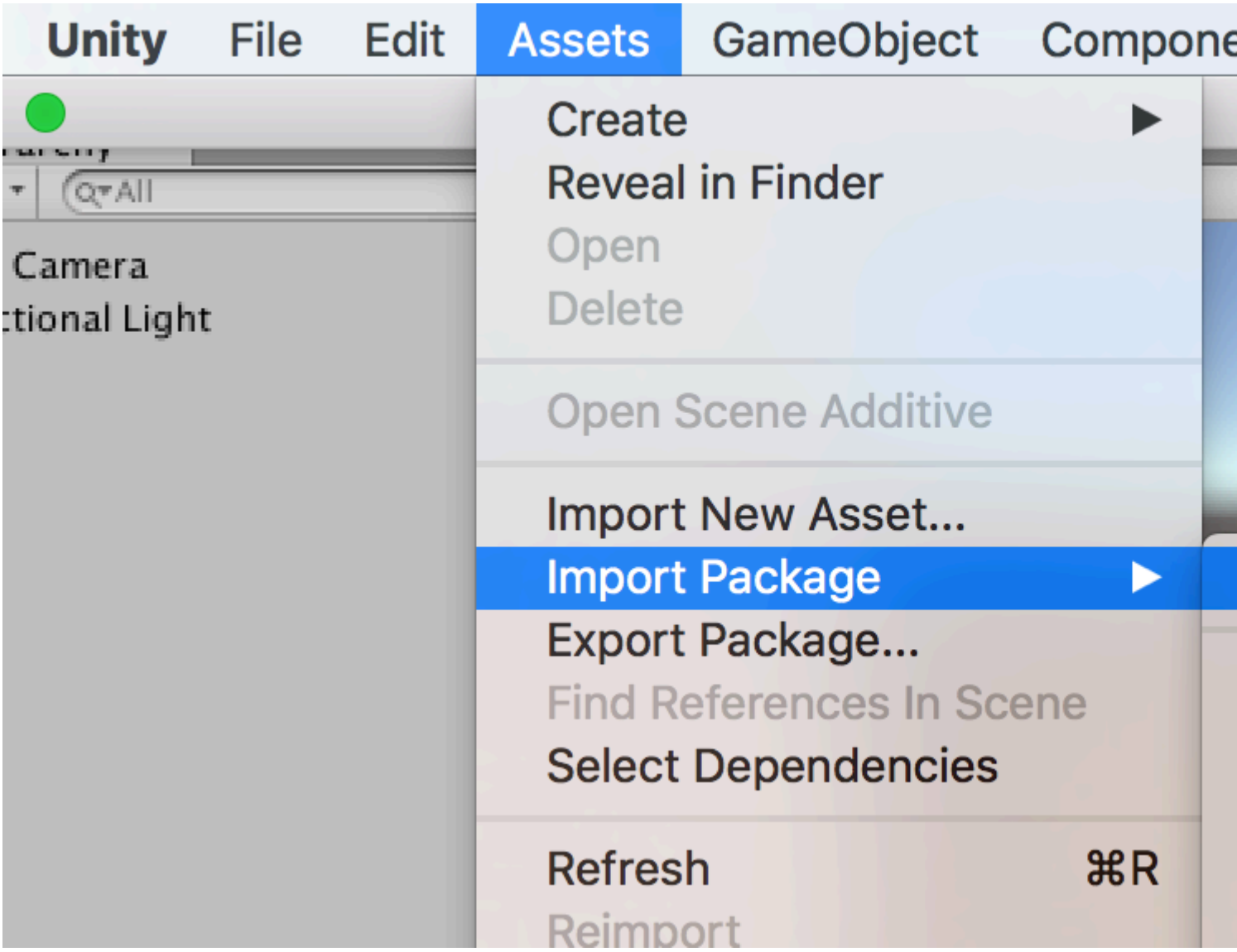
Examples

Asset packages

Asset Packages (with the file format of `.unitypackage`) are a commonly used way of distributing Unity projects to other users. When working with peripherals that have their own SDKs (eg. Oculus), you may be asked to download and import one of these packages.

Importing a .unitypackage

To import a package, go to the Unity menu bar and click on `Assets > Import Package > Custom Package...`, then navigate to the `.unitypackage` file in the File Browser that appears.



Read How to use asset packages online: https://riptutorial.com/unity3d/topic/4491/how-to-use-asset-packages

Chapter 15: Immediate Mode Graphical User Interface System (IMGUI)

Syntax

- public static void GUILayout.Label(string text, params GUILayoutOption[] options)
- public static bool GUILayout.Button(string text, params GUILayoutOption[] options)
- public static string GUILayout.TextArea(string text, params GUILayoutOption[] options)

Examples

GUILayout

Old UI system tool, now used for fast and simple prototyping or debugging in game.

```
void OnGUI ()
{
    GUILayout.Label ("I'm a simple label text displayed in game.");

    if ( GUILayout.Button("CLICK ME") )
    {
        GUILayout.TextArea ("This is a \n
                             multiline comment.")
    }
}
```

GUILayout function works inside the **OnGUI** function.

Read Immediate Mode Graphical User Interface System (IMGUI) online:
https://riptutorial.com/unity3d/topic/6947/immediate-mode-graphical-user-interface-system--imgui-

Chapter 16: Importers and (Post)Processors

Syntax

- AssetPostprocessor.OnPreprocessTexture()

Remarks

Use `String.Contains()` to process only assets that have a given string in their asset paths.

```
if (assetPath.Contains("ProcessThisFolder"))
{
    // Process asset
}
```

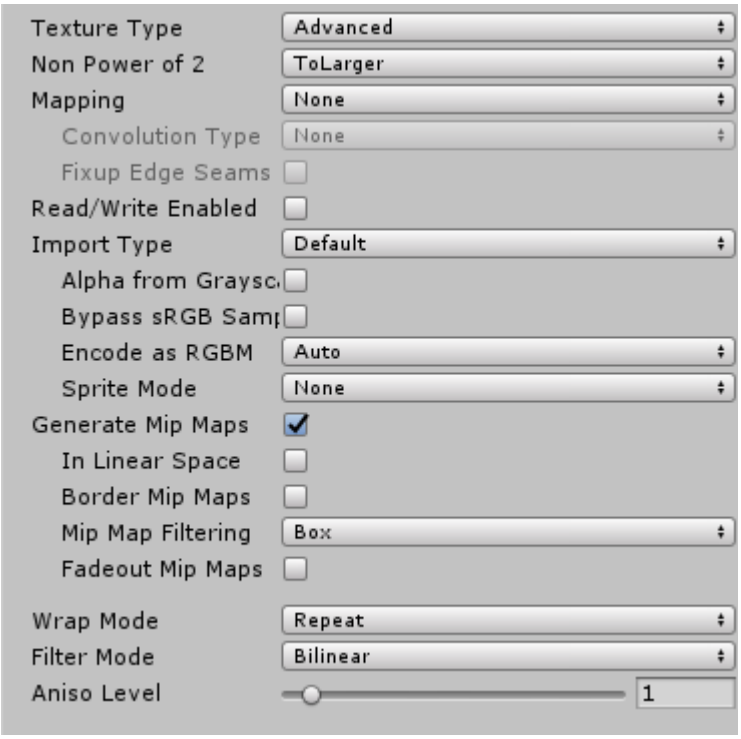
Examples

Texture postprocessor

Create `TexturePostProcessor.cs` file anywhere in **Assets** folder:

```
using UnityEngine;
using UnityEditor;

public class TexturePostProcessor : AssetPostprocessor
{
    void OnPostprocessTexture(Texture2D texture)
    {
        TextureImporter importer = assetImporter as TextureImporter;
        importer.anisoLevel = 1;
        importer.filterMode = FilterMode.Bilinear;
        importer.mipmapEnabled = true;
        importer.npotScale = TextureImporterNPOTScale.ToLarger;
        importer.textureType = TextureImporterType.Advanced;
    }
}
```

Now, every time Unity imports a texture it will have the following parameters:



If you use postprocessor, you can not change texture parameters by manipulating **Import Settings** in editor.
When you hit **Apply** button the texture will be reimported and postprocessor code will run again.

A Basic Importer

Assume you have a custom file you want to create an importer for. It could be an .xls file or whatever. In this case we're going to use a JSON file because it's easy but we're going to pick a custom extension to make it easy to tell which files are ours?

Let's assume the format of the JSON file is

```
{
  "someValue": 123,
  "someOtherValue": 456.297,
  "someBoolValue": true,
  "someStringValue": "this is a string",
}
```

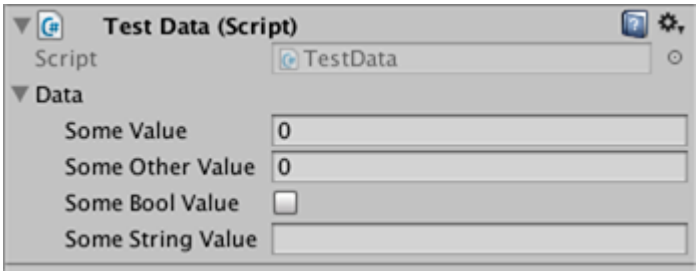
Let's save that as `Example.test` somewhere *outside* of assets for now.

Next make a `MonoBehaviour` with a custom class just for the data. The custom class is solely to make it easy to deserialize the JSON. You do NOT have to use a custom class but it makes this example shorter. We'll save this in `TestData.cs`

```
using UnityEngine;
using System.Collections;

public class TestData : MonoBehaviour {
```

```
    [System.Serializable]
    public class Data {
        public int someValue = 0;
        public float someOtherValue = 0.0f;
        public bool someBoolValue = false;
        public string someStringValue = "";
    }

    public Data data = new Data();
}
```

If you were to manually add that script to a GameObject you'd see something like



Next make an `Editor` folder somewhere under `Assets`. I can be at any level. Inside the Editor folder make a `TestDataAssetPostprocessor.cs` file and put this in it.

```
using UnityEditor;
using UnityEngine;
using System.Collections;

public class TestDataAssetPostprocessor : AssetPostprocessor
{
    const string s_extension = ".test";

    // NOTE: Paths start with "Assets/"
    static bool IsFileWeCareAbout(string path)
    {
        return System.IO.Path.GetExtension(path).Equals(
           s_extension,
           System.StringComparison.Ordinal);
    }

    static void HandleAddedOrChangedFile(string path)
    {
        string text = System.IO.File.ReadAllText(path);
        // should we check for error if the file can't be parsed?
        TestData.Data newData = JsonUtility.FromJson<TestData.Data>(text);

        string prefabPath = path + ".prefab";
        // Get the existing prefab
        GameObject existingPrefab =
            AssetDatabase.LoadAssetAtPath(prefabPath, typeof(Object)) as GameObject;
        if (!existingPrefab)
        {
            // If no prefab exists make one
            GameObject newGameObject = new GameObject();
            newGameObject.AddComponent<TestData>();
            PrefabUtility.CreatePrefab(prefabPath,
                                       newGameObject,
                                       ReplacePrefabOptions.Default);
```

```
            GameObject.DestroyImmediate(newGameObject);
            existingPrefab =
                AssetDatabase.LoadAssetAtPath(prefabPath, typeof(Object)) as GameObject;
        }

        TestData testData = existingPrefab.GetComponent<TestData>();
        if (testData != null)
        {
            testData.data = newData;
            EditorUtility.SetDirty(existingPrefab);
        }
    }

    static void HandleRemovedFile(string path)
    {
        // Decide what you want to do here. If the source file is removed
        // do you want to delete the prefab? Maybe ask if you'd like to
        // remove the prefab?
        // NOTE: Because you might get many calls (like you deleted a
        // subfolder full of .test files you might want to get all the
        // filenames and ask all at once ("delete all these prefabs?").
    }

    static void OnPostprocessAllAssets (string[] importedAssets, string[] deletedAssets,
string[] movedAssets, string[] movedFromAssetPaths)
    {
        foreach (var path in importedAssets)
        {
            if (IsFileWeCareAbout(path))
            {
                HandleAddedOrChangedFile(path);
            }
        }

        foreach (var path in deletedAssets)
        {
            if (IsFileWeCareAbout(path))
            {
                HandleRemovedFile(path);
            }
        }

        for (var ii = 0; ii < movedAssets.Length; ++ii)
        {
            string srcStr = movedFromAssetPaths[ii];
            string dstStr = movedAssets[ii];

            // the source was moved, let's move the corresponding prefab
            // NOTE: We don't handle the case if there already being
            // a prefab of the same name at the destination
            string srcPrefabPath = srcStr + ".prefab";
            string dstPrefabPath = dstStr + ".prefab";

            AssetDatabase.MoveAsset(srcPrefabPath, dstPrefabPath);
        }
    }
}
```

With that saved you should be able to drag and drop the `Example.test` file we created above into your Unity Assets folder and you should see the corresponding prefab created. If you edit

`Example.test` you'll see the data in the prefab is updated immediately. If you drag the prefab into the scene hierarchy you'll see it update as well as `Example.test` changes. If you move `Example.test` to another folder the corresponding prefab will move with it. If you change a field on an instance then change the `Example.test` file you'll see only the fields you didn't modify on the instance get updated.

Improvements: In the example above, after you drag `Example.test` into your `Assets` folder you'll see there's both an `Example.test` and an `Example.test.prefab`. It would be great to know to make it work more like the model importers work we're you'd magically only see `Example.test` and it's an `AssetBundle` or some such thing. If you know how please provide that example

Read Importers and (Post)Processors online: https://riptutorial.com/unity3d/topic/5279/importers-and--post-processors

Chapter 17: Input System

Examples

Reading Key Press and difference between GetKey, GetKeyDown and GetKeyUp

Input must must read from the Update function.

Reference for all the available Keycode enum.

1. Reading key press with `Input.GetKey`:

`Input.GetKey` will **repeatedly** return `true` while the user holds down the specified key. This can be used to **repeatedly** fire a weapon while holding the specified key down. Below is an example of bullet auto-fire when the Space key is held down. The player doesn't have to press and release the key over and over again.

```
public GameObject bulletPrefab;
public float shootForce = 50f;

void Update()
{
    if (Input.GetKey(KeyCode.Space))
    {
        Debug.Log("Shooting a bullet while SpaceBar is held down");

        //Instantiate bullet
        GameObject bullet = Instantiate(bulletPrefab, transform.position, transform.rotation)
as GameObject;

        //Get the Rigidbody from the bullet then add a force to the bullet
        bullet.GetComponent<Rigidbody>().AddForce(bullet.transform.forward * shootForce);
    }
}
```

2.Reading key press with `Input.GetKeyDown`:

`Input.GetKeyDown` will true only **once** when the specified key is pressed. This is the key difference between `Input.GetKey` and `Input.GetKeyDown`. One example use of its use is to toggle a UI or flashlight or an item on/off.

```
public Light flashLight;
bool enableFlashLight = false;

void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        //Toggle Light
        enableFlashLight = !enableFlashLight;
```

```
        if (enableFlashLight)
        {
            flashLight.enabled = true;
            Debug.Log("Light Enabled!");
        }
        else
        {
            flashLight.enabled = false;
            Debug.Log("Light Disabled!");
        }
    }
}
```

3.Reading key press with `Input.GetKeyUp`:

This is the exact opposite of `Input.GetKeyDown`. It is used to detect when key-press is released/lifted. Just like `Input.GetKeyDown`, it returns `true` only **once**. For example, you can `enable` light when key is held down with `Input.GetKeyDown` then disable the light when key is released with `Input.GetKeyUp`.

```
public Light flashLight;
void Update()
{
    //Disable Light when Space Key is pressed
    if (Input.GetKeyDown(KeyCode.Space))
    {
        flashLight.enabled = true;
        Debug.Log("Light Enabled!");
    }

    //Disable Light when Space Key is released
    if (Input.GetKeyUp(KeyCode.Space))
    {
        flashLight.enabled = false;
        Debug.Log("Light Disabled!");
    }
}
```

Read Accelerometer Sensor (Basic)

`Input.acceleration` is used to read the accelerometer sensor. It returns `Vector3` as a result which contains `x`,`y` and `z` axis values in 3D space.

```
void Update()
{
    Vector3 acclerometerValue = rawAccelValue();
    Debug.Log("X: " + acclerometerValue.x + "  Y: " + acclerometerValue.y + "  Z: " +
acclerometerValue.z);
}

Vector3 rawAccelValue()
{
    return Input.acceleration;
}
```

Read Accelerometer Sensor (Advance)

Using raw values directly from the accelerometer sensor to move or rotate a GameObject can cause problems such as jerky movements or vibrations. It is recommended to smooth out the values before using them. In fact, values from the accelerometer sensor should always be smoothed out before use. This can be accomplished with a low pass filter and this is where `Vector3.Lerp` comes into place.

```
//The lower this value, the less smooth the value is and faster Accel is updated. 30 seems
fine for this
const float updateSpeed = 30.0f;

float AccelerometerUpdateInterval = 1.0f / updateSpeed;
float LowPassKernelWidthInSeconds = 1.0f;
float LowPassFilterFactor = 0;
Vector3 lowPassValue = Vector3.zero;

void Start()
{
    //Filter Accelerometer
    LowPassFilterFactor = AccelerometerUpdateInterval / LowPassKernelWidthInSeconds;
    lowPassValue = Input.acceleration;
}

void Update()
{

    //Get Raw Accelerometer values (pass in false to get raw Accelerometer values)
    Vector3 rawAccelValue = filterAccelValue(false);
    Debug.Log("RAW X: " + rawAccelValue.x + "  Y: " + rawAccelValue.y + "  Z: " +
rawAccelValue.z);

    //Get smoothed Accelerometer values (pass in true to get Filtered Accelerometer values)
    Vector3 filteredAccelValue = filterAccelValue(true);
    Debug.Log("FILTERED X: " + filteredAccelValue.x + "  Y: " + filteredAccelValue.y + "  Z: "
+ filteredAccelValue.z);
}

//Filter Accelerometer
Vector3 filterAccelValue(bool smooth)
{
    if (smooth)
        lowPassValue = Vector3.Lerp(lowPassValue, Input.acceleration, LowPassFilterFactor);
    else
        lowPassValue = Input.acceleration;

    return lowPassValue;
}
```

Read Accelerometer Sensor(Precision)

Read the accelerometer Sensor with precision.

This example allocates memory:

```
void Update()
```

```
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
    Debug.Log("PRECISE X: " + accelValue.x + "  Y: " + accelValue.y + "  Z: " + accelValue.z);

}

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    foreach (AccelerationEvent tempAccelEvent in Input.accelerationEvents)
    {
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}
```

This example does **not** allocates memory:

```
void Update()
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
    Debug.Log("PRECISE X: " + accelValue.x + "  Y: " + accelValue.y + "  Z: " + accelValue.z);

}

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    for (int i = 0; i < Input.accelerationEventCount; ++i)
    {
        AccelerationEvent tempAccelEvent = Input.GetAccelerationEvent(i);
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}
```

Note that this is not filtered. Please look here for how to smooth accelerometer values to remove noise.

Read Mouse Button (Left, Middle, Right) Clicks

These functions are used to check Mouse Button Clicks.

- `Input.GetMouseButton(int button);`
- `Input.GetMouseButtonDown(int button);`
- `Input.GetMouseButtonUp(int button);`

They all take the-same parameter.

- 0 = Left Mouse Click.
- 1 = Right Mouse Click.
- 2 = Middle Mouse Click.

`GetMouseButton` is used to detect when mouse button is *continuously held* down. It returns `true`

while the specified mouse button is being held down.

```
void Update()
{
    if (Input.GetMouseButton(0))
    {
        Debug.Log("Left Mouse Button Down");
    }

    if (Input.GetMouseButton(1))
    {
        Debug.Log("Right Mouse Button Down");
    }

    if (Input.GetMouseButton(2))
    {
        Debug.Log("Middle Mouse Button Down");
    }
}
```

`GetMouseButtonDown` is used to detect when there is mouse click. It returns `true` if it is pressed **once**. It won't return `true` again until the mouse button is released and pressed again.

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Debug.Log("Left Mouse Button Clicked");
    }

    if (Input.GetMouseButtonDown(1))
    {
        Debug.Log("Right Mouse Button Clicked");
    }

    if (Input.GetMouseButtonDown(2))
    {
        Debug.Log("Middle Mouse Button Clicked");
    }
}
```

`GetMouseButtonUp` is used to detect when the specififed mouse button is released. This is will only return `true` once the specified mouse button is released. To return true again, it has to be pressed and released again.

```
void Update()
{
    if (Input.GetMouseButtonUp(0))
    {
        Debug.Log("Left Mouse Button Released");
    }

    if (Input.GetMouseButtonUp(1))
    {
        Debug.Log("Right Mouse Button Released");
    }
```

```
    if (Input.GetMouseButtonUp(2))
    {
        Debug.Log("Middle Mouse Button Released");
    }
}
```

Read Input System online: https://riptutorial.com/unity3d/topic/3413/input-system

Chapter 18: Layers

Examples

Layer usage

Unity layers are similar to tags as in that they can be used to define objects that should be interacted with or should behave in a certain manner, however, layers are mainly used with functions in the `Physics` class: Unity Documentation - Physics

Layers are represented by an integer and can be passed to the functions in this manner:

```
using UnityEngine;
class LayerExample {

    public int layer;

    void Start()
    {
        Collider[] colliders = Physics.OverlapSphere(transform.position, 5f, layer);
    }
}
```

Using a layer in this manner will include only Colliders whose GameObjects have the layer specified in the calculations done. This makes further logic simpler as well as improving performance.

LayerMask Structure

The `LayerMask` structure is an interface that functions almost exactly like passing an integer to the function in question. However, its biggest benefit is allowing the user to select the layer in question from a drop-down menu in the inspector.

```
using UnityEngine;
class LayerMaskExample{

    public LayerMask mask;
    public Vector3 direction;

    void Start()
    {
        if(Physics.Raycast(transform.position, direction, 35f, mask))
        {
            Debug.Log("Raycast hit");
        }
    {
}
```

It also has multiple static functions that allow for converting layer names to indices or indices to layer names.

```
using UnityEngine;
class NameToLayerExample{

    void Start()
    {
        int layerindex = LayerMask.NameToLayer("Obstacle");
    {
}
```

In order to make Layer checking easy define the following extension method.

```
public static bool IsInLayerMask(this GameObject @object, LayerMask layerMask)
{
    bool result = (1 << @object.layer & layerMask) == 0;

    return result;
}
```

This method will allow you to check whether a gameobject is in a layermask (selected in the editor) or not.

Read Layers online: https://riptutorial.com/unity3d/topic/4762/layers

Chapter 19: Mobile platforms

Syntax

- public static int Input.touchCount
- public static Touch Input.GetTouch(int index)

Examples

Detecting Touch

To detect a touch in Unity it's quite simple we just have to use `Input.GetTouch()` and pass it an index.

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        if (Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Began)
        {
            //Do Stuff
        }
    }
}
```

or

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        for(int i = 0; i < Input.touchCount; i++)
        {
            if (Input.GetTouch(i).phase == TouchPhase.Began)
            {
                //Do Stuff
            }
        }
    }
}
```

These examples gets the touch of the last game frame.

TouchPhase

Inside of the TouchPhase enum there are 5 different kind of TouchPhase's

- Began - a finger touched the screen
- Moved - a finger moved on the screen
- Stationary - a finger is on the screen but is not moving
- Ended - a finger was lifted from the screen
- Canceled - the system cancelled tracking for the touch

For example to move the object this script is attached to across the screen based on touch.

```
public class TouchMoveExample : MonoBehaviour
{
    public float speed = 0.1f;

    void Update () {
        if(Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Moved)
        {
            Vector2 touchDeltaPosition = Input.GetTouch(0).deltaPosition;
            transform.Translate(-touchDeltaPosition.x * speed, -touchDeltaPosition.y * speed,
0);
        }
    }
}
```

Read Mobile platforms online: https://riptutorial.com/unity3d/topic/6285/mobile-platforms

Chapter 20: MonoBehaviour class implementation

Examples

No overridden methods

The reason you do not have to override `Awake`, `Start`, `Update` and other method is because they are not virtual methods defined in a base class.

The first time your script gets accessed, the scripting runtime looks through the script to see if some methods are defined. If they are, that information is cached and the methods are added to their respective list. These lists are then simply looped through at different times.

The reason these methods are not virtual is because of performance. If all the scripts would have `Awake`, `Start`, `OnEnable`, `OnDisable`, `Update`, `LateUpdate`, and `FixedUpdate`, then these would all be added to their lists what would mean that all these methods get executed. Normally this wouldn't be a big problem, however, all these method calls are from the native side (C++) to the managed side (C#) which comes with a performance cost.

Now imagine this, all these methods are in their lists and some/most of them might not even have an actual method body. This would mean that there is a huge amount of performance wasted on calling methods that do not even do anything. To prevent this, Unity opted out of using virtual methods and made a messaging system that makes sure that these methods only get called when they are actually defined, saving unnecessary method calls.

You can read more on the matter on an Unity blog over here: 10000 Update() Calls and more on IL2CPP over here: An Introduction to IL2CPP Internals

Read MonoBehaviour class implementation online:
https://riptutorial.com/unity3d/topic/2304/monobehaviour-class-implementation

Chapter 21: Multiplatform development

Examples

Compiler Definitions

Compiler definitions run platform specific code. Using them you can make small differences between various platforms.

- Trigger Game Center achievements on apple devices and google play achievements on Android devices.
- Change the icons in menus (windows logo in windows, Linux penguin in Linux).
- Possibly have platform specific mechanics depending on the platform.
- And much more...

```
void Update(){

#if UNITY_IPHONE
    //code here is only called when running on iPhone
#endif

#if UNITY_STANDALONE_WIN && !UNITY_EDITOR
    //code here is only ran in a unity game running on windows outside of the editor
#endif

//other code that will be ran regardless of platform

}
```

A complete list of Unity compiler definitions can be found here

Organizing platform specific methods to partial classes

Partial classes provide a clean way to separate core logic of your scripts from platform specific methods.

Partial classes and methods are marked with the keyword `partial`. This signals the compiler to leave the class "open" and look in other files for the rest of the implementation.

```
// ExampleClass.cs
using UnityEngine;

public partial class ExampleClass : MonoBehaviour
{
    partial void PlatformSpecificMethod();

    void OnEnable()
    {
        PlatformSpecificMethod();
    }
}
```

Now we can create files for our platform specific scripts that implement the partial method. Partial methods can have parameters (also `ref`) but must return `void`.

```
// ExampleClass.Iphone.cs

#if UNITY_IPHONE
using UnityEngine;

public partial class ExampleClass
{
    partial void PlatformSpecificMethod()
    {
        Debug.Log("I am an iPhone");
    }
}
#endif
```

```
// ExampleClass.Android.cs

#if UNITY_ANDROID
using UnityEngine;

public partial class ExampleClass
{
    partial void PlatformSpecificMethod()
    {
        Debug.Log("I am an Android");
    }
}
#endif
```

If a partial method is not implemented, the compiler will omit the call.

> Tip: This pattern is useful when creating Editor specific methods as well.

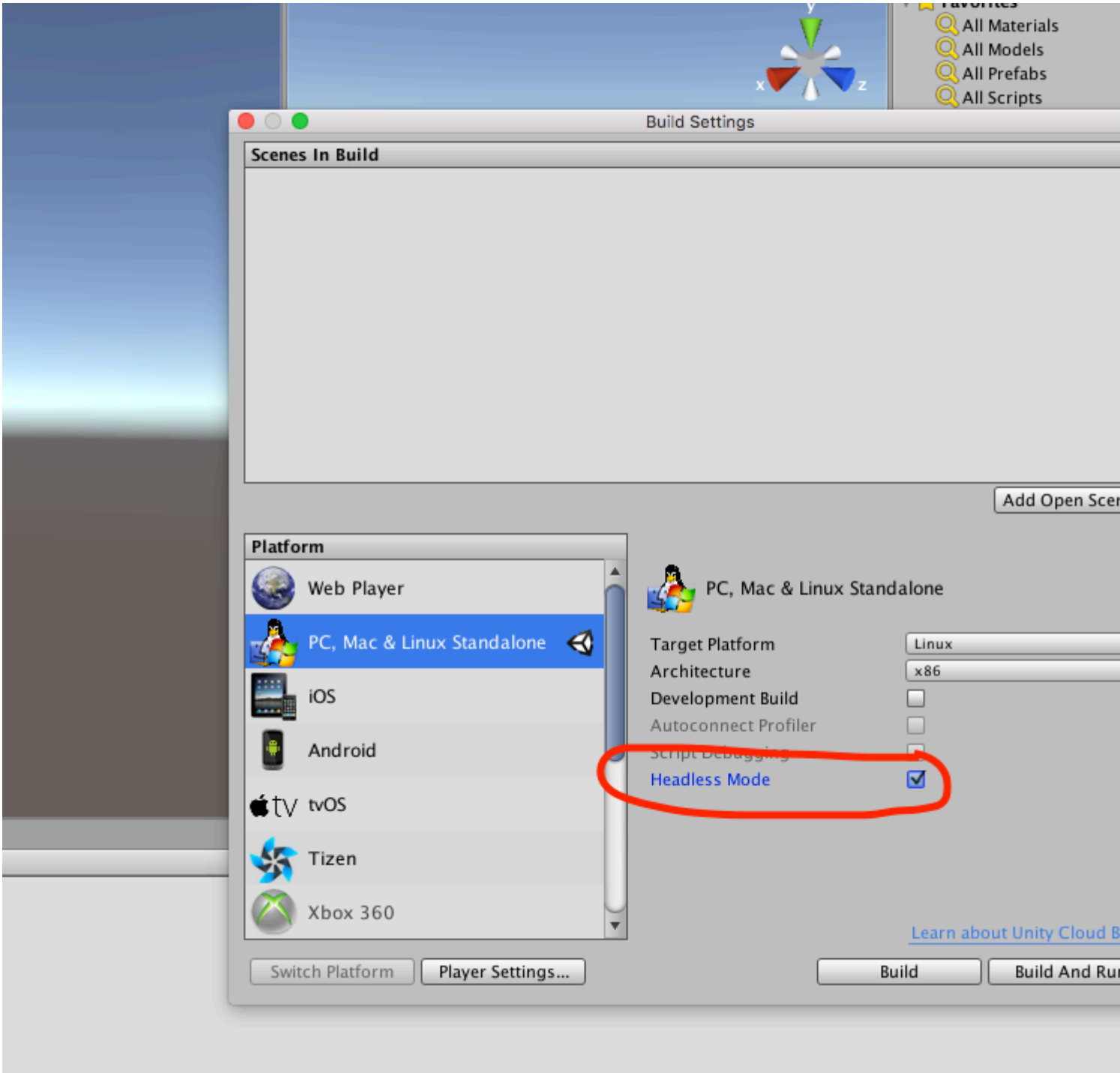
Read Multiplatform development online: https://riptutorial.com/unity3d/topic/4816/multiplatform-development

Chapter 22: Networking

Remarks

Headless mode in Unity

If you are building a Server to deploy in Linux, the Build settings have a "Headless mode" option. An application build with this option doesn't display anything and doesn't read user input, which is usually what we want for a Server.



Examples

Creating a server, a client, and sending a message.

Unity networking provides the High Level API (HLA) to handle network communications abstracting from low level implementations.

In this example we will see how to create a Server that can communicate with one or multiple clients.

The HLA allows us to easily serialize a class and send objects of this class over the network.

The Class we are using to serialize

This class have to inherance from MessageBase, in this example we will just send a string inside this class.

```
using System;
using UnityEngine.Networking;

public class MyNetworkMessage : MessageBase
{
    public string message;
}
```

Creating a Server

We create a server that listen to the port 9999, allows a maximum of 10 connections, and read objects from the network of our custom class.

The HLA associates different types of message to an id. There are default messages type defined in the MsgType class from Unity Networking. For example the connect type have id 32 and it is called in the server when a client connects to it, or in the client when it connects to a server. You can register handlers to manage the different types of message.

When you are sending a custom class, like our case, we define a handlers with a new id associated to the class we are sending over the network.

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;

public class Server : MonoBehaviour {

    int port = 9999;
    int maxConnections = 10;

    // The id we use to identify our messages and register the handler
```

```
short messageID = 1000;

// Use this for initialization
void Start () {
    // Usually the server doesn't need to draw anything on the screen
    Application.runInBackground = true;
    CreateServer();
}

void CreateServer() {
    // Register handlers for the types of messages we can receive
    RegisterHandlers ();

    var config = new ConnectionConfig ();
    // There are different types of channels you can use, check the official documentation
    config.AddChannel (QosType.ReliableFragmented);
    config.AddChannel (QosType.UnreliableFragmented);

    var ht = new HostTopology (config, maxConnections);

    if (!NetworkServer.Configure (ht)) {
        Debug.Log ("No server created, error on the configuration definition");
        return;
    } else {
        // Start listening on the defined port
        if(NetworkServer.Listen (port))
            Debug.Log ("Server created, listening on port: " + port);
        else
            Debug.Log ("No server created, could not listen to the port: " + port);
    }
}

void OnApplicationQuit() {
    NetworkServer.Shutdown ();
}

private void RegisterHandlers () {
    // Unity have different Messages types defined in MsgType
    NetworkServer.RegisterHandler (MsgType.Connect, OnClientConnected);
    NetworkServer.RegisterHandler (MsgType.Disconnect, OnClientDisconnected);

    // Our message use his own message type.
    NetworkServer.RegisterHandler (messageID, OnMessageReceived);
}

private void RegisterHandler(short t, NetworkMessageDelegate handler) {
    NetworkServer.RegisterHandler (t, handler);
}

void OnClientConnected(NetworkMessage netMessage)
{
    // Do stuff when a client connects to this server

    // Send a thank you message to the client that just connected
    MyNetworkMessage messageContainer = new MyNetworkMessage();
    messageContainer.message = "Thanks for joining!";

    // This sends a message to a specific client, using the connectionId
    NetworkServer.SendToClient(netMessage.conn.connectionId,messageID,messageContainer);

    // Send a message to all the clients connected
```

```
        messageContainer = new MyNetworkMessage();
        messageContainer.message = "A new player has conencted to the server";

        // Broadcast a message a to everyone connected
        NetworkServer.SendToAll(messageID,messageContainer);
    }

    void OnClientDisconnected(NetworkMessage netMessage)
    {
        // Do stuff when a client dissconnects
    }

    void OnMessageReceived(NetworkMessage netMessage)
    {
        // You can send any object that inherence from MessageBase
        // The client and server can be on different projects, as long as the MyNetworkMessage
or the class you are using have the same implementation on both projects
        // The first thing we do is deserialize the message to our custom type
        var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();
        Debug.Log("Message received: " + objectMessage.message);

    }
}
```

The Client

Now we create a Client

```
using System;
using UnityEngine;
using UnityEngine.Networking;


public class Client : MonoBehaviour
{
    int port = 9999;
    string ip = "localhost";

    // The id we use to identify our messages and register the handler
    short messageID = 1000;

    // The network client
    NetworkClient client;

    public Client ()
    {
        CreateClient();
    }

    void CreateClient()
    {
        var config = new ConnectionConfig ();

        // Config the Channels we will use
        config.AddChannel (QosType.ReliableFragmented);
        config.AddChannel (QosType.UnreliableFragmented);

        // Create the client ant attach the configuration
```

```
        client = new NetworkClient ();
        client.Configure (config,1);

        // Register the handlers for the different network messages
        RegisterHandlers();

        // Connect to the server
        client.Connect (ip, port);
    }

    // Register the handlers for the different message types
    void RegisterHandlers () {

        // Unity have different Messages types defined in MsgType
        client.RegisterHandler (messageID, OnMessageReceived);
        client.RegisterHandler(MsgType.Connect, OnConnected);
        client.RegisterHandler(MsgType.Disconnect, OnDisconnected);
    }

    void OnConnected(NetworkMessage message) {
        // Do stuff when connected to the server

        MyNetworkMessage messageContainer = new MyNetworkMessage();
        messageContainer.message = "Hello server!";

        // Say hi to the server when connected
        client.Send(messageID,messageContainer);
    }

    void OnDisconnected(NetworkMessage message) {
        // Do stuff when disconnected to the server
    }

    // Message received from the server
    void OnMessageReceived(NetworkMessage netMessage)
    {
        // You can send any object that inherence from MessageBase
        // The client and server can be on different projects, as long as the MyNetworkMessage
or the class you are using have the same implementation on both projects
        // The first thing we do is deserialize the message to our custom type
        var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();

        Debug.Log("Message received: " + objectMessage.message);
    }
}
```

Read Networking online: https://riptutorial.com/unity3d/topic/5671/networking

Chapter 23: Object Pooling

Examples

Object Pool

Sometimes when you make a game you need to create and destroy a lot of objects of the same type over and over again. You can simply do this by making a prefab and instantiate/destroy this whenever you need to, however, doing this is inefficient and can slow your game down.

One way to get around this issue is object pooling. Basically what this means is that you have a pool (with or without a limit to the amount) of objects that you are going to reuse whenever you can to prevent unnecessary instantiating or destroying.

Below is an example of a simple object pool

```
public class ObjectPool : MonoBehaviour
{
    public GameObject prefab;
    public int amount = 0;
    public bool populateOnStart = true;
    public bool growOverAmount = true;

    private List<GameObject> pool = new List<GameObject>();

    void Start()
    {
        if (populateOnStart && prefab != null && amount > 0)
        {
            for (int i = 0; i < amount; i++)
            {
                var instance = Instantiate(Prefab);
                instance.SetActive(false);
                pool.Add(instance);
            }
        }
    }

    public GameObject Instantiate (Vector3 position, Quaternion rotation)
    {
        foreach (var item in pool)
        {
            if (!item.activeInHierarchy)
            {
                item.transform.position = position;
                item.transform.rotation = rotation;
                item.SetActive( true );
                return item;
            }
        }

        if (growOverAmount)
        {
            var instance = (GameObject)Instantiate(prefab, position, rotation);
```

```
            pool.Add(instance);
            return instance;
        }

        return null;
    }
}
```

Let's go over the variables first

```
public GameObject prefab;
public int amount = 0;
public bool populateOnStart = true;
public bool growOverAmount = true;

private List<GameObject> pool = new List<GameObject>();
```

- `GameObject prefab`: this is the prefab that the object pool will use to instantiate new objects into the pool.
- `int amount`: This is the maximum amount of items that can be in the pool. If you want to instantiate another item and the pool has already reached its limit then another item from the pool will be used.
- `bool populateOnStart`: you can choose to populate the pool on start or not. Doing so will fill up the pool with instances of the prefab so that the first time you call `Instantiate` you will get an already existing object
- `bool growOverAmount`: Setting this to true allows the pool to grow whenever more than the amount are requested in a certain timeframe. You are not always able to accurately predict the amount of items to put in your pool so this will add more to your pool when needed.
- `List<GameObject> pool`: this is the pool, the place where all your instantiated/destroyed objects are stored.

Now let's check out the `Start` function

```
void Start()
{
    if (populateOnStart && prefab != null && amount > 0)
    {
        for (int i = 0; i < amount; i++)
        {
            var instance = Instantiate(Prefab);
            instance.SetActive(false);
            pool.Add(instance);
        }
    }
}
```

In the start function we check if we should populate the list on start and do so if the `prefab` has been set and the amount is bigger than 0 (otherwise we would be creating indefinitely).

This is just a simple for loop instantiating new objects and putting them in the pool. One thing to pay attention to is that we set all the instances to inactive. This way they are not visible in the game yet.

Next, there is the `Instantiate` function, which is where most of the magic happens

```
public GameObject Instantiate (Vector3 position, Quaternion rotation)
{
    foreach (var item in pool)
    {
        if (!item.activeInHierarchy)
        {
            item.transform.position = position;
            item.transform.rotation = rotation;
            item.SetActive(true);
            return item;
        }
    }

    if (growOverAmount)
    {
        var instance = (GameObject)Instantiate(prefab, position, rotation);
        pool.Add(instance);
        return instance;
    }

    return null;
}
```

The `Instantiate` function looks just like Unity's own `Instantiate` function, except the prefab has already been provided above as a class member.

The first step of the `Instantiate` function is checking to see if there is an inactive object in the pool right now. This means that we can reuse that object and give it back to the requester. If there is an inactive object in the pool we set the position and the rotation, set it to be active (otherwise it could be reused by accident if you forget to activate it) and return it to the requester.

The second step only happens if there are no inactive items in the pool and the pool is allowed to grow over the initial amount. What happens is simple: another instance of the prefab is created and added to the pool. Allowing growth of the pool helps you in having the right amount of objects in the pool.

The third "step" only happens if there are no inactive items in the pool and the pool is *not* allowed to grow. When this happens the requester will receive a null GameObject which means that nothing was available and should be handled properly to prevent `NullReferenceExceptions`.

Important!

To make sure your items get back into the pool you should **not** destroy the game objects. The only thing you need to do is set them to inactive and that will make them available for reusage through the pool.

Simple object pool

Below is an example of an object pool that allows renting and returning of a given object type. To create the object pool a Func for the create function and an Action to destroy the object are required to give the user flexibility. On requesting an object when the pool is empty a new object

will be created and on requesting when the pool has objects then objects are removed from the pool and returned.

Object Pool

```
public class ResourcePool<T> where T : class
{
    private readonly List<T> objectPool = new List<T>();
    private readonly Action<T> cleanUpAction;
    private readonly Func<T> createAction;

    public ResourcePool(Action<T> cleanUpAction, Func<T> createAction)
    {
        this.cleanUpAction = cleanUpAction;
        this.createAction = createAction;
    }

    public void Return(T resource)
    {
        this.objectPool.Add(resource);
    }

    private void PurgeSingleResource()
    {
        var resource = this.Rent();
        this.cleanUpAction(resource);
    }

    public void TrimResourcesBy(int count)
    {
        count = Math.Min(count, this.objectPool.Count);
        for (int i = 0; i < count; i++)
        {
            this.PurgeSingleResource();
        }
    }

    public T Rent()
    {
        int count = this.objectPool.Count;
        if (count == 0)
        {
            Debug.Log("Creating new object.");
            return this.createAction();
        }
        else
        {
            Debug.Log("Retrieving existing object.");
            T resource = this.objectPool[count-1];
            this.objectPool.RemoveAt(count-1);
            return resource;
        }
    }
}
```

Sample usage

```
public class Test : MonoBehaviour
{
```

```
    private ResourcePool<GameObject> objectPool;

    [SerializeField]
    private GameObject enemyPrefab;

    void Start()
    {
        this.objectPool = new ResourcePool<GameObject>(Destroy,() =>
Instantiate(this.enemyPrefab) );
    }

    void Update()
    {
        // To get existing object or create new from pool
        var newEnemy = this.objectPool.Rent();
        // To return object to pool
        this.objectPool.Return(newEnemy);
        // In this example the message 'Creating new object' should only be seen on the frame
call
        // after that the same object in the pool will be returned.
    }
}
```

Another simple object pool

Another example: a Weapon that shoots out Bullets.

The Weapon acts as an object pool for the Bullets it creates.

```
public class Weapon : MonoBehaviour {

    // The Bullet prefab that the Weapon will create
    public Bullet bulletPrefab;

    // This List is our object pool, which starts out empty
    private List<Bullet> availableBullets = new List<Bullet>();

    // The Transform that will act as the Bullet starting position
    public Transform bulletInstantiationPoint;

    // To spawn a new Bullet, this method either grabs an available Bullet from the pool,
    // otherwise Instantiates a new Bullet
    public Bullet CreateBullet () {
        Bullet newBullet = null;

        // If a Bullet is available in the pool, take the first one and make it active
        if (availableBullets.Count > 0) {
            newBullet = availableBullets[availableBullets.Count - 1];

            // Remove the Bullet from the pool
            availableBullets.RemoveAt(availableBullets.Count - 1);

            // Set the Bullet's position and make its GameObject active
            newBullet.transform.position = bulletInstantiationPoint.position;
            newBullet.gameObject.SetActive(true);
        }
        // If no Bullets are available in the pool, Instantiate a new Bullet
        else {
            newBullet newObject = Instantiate(bulletPrefab, bulletInstantiationPoint.position,
```

```
Quaternion.identity);

            // Set the Bullet's Weapon so we know which pool to return to later on
            newBullet.weapon = this;
        }

        return newBullet;
    }

}

public class Bullet : MonoBehaviour {

    public Weapon weapon;

    // When Bullet collides with something, rather than Destroying it, we return it to the
pool
    public void ReturnToPool () {
        // Add Bullet to the pool
        weapon.availableBullets.Add(this);

        // Disable the Bullet's GameObject so it's hidden from view
        gameObject.SetActive(false);
    }

}
```

Read Object Pooling online: https://riptutorial.com/unity3d/topic/2276/object-pooling

Chapter 24: Optimization

Remarks

1. If possible, disable scripts on objects when they are not needed. For example if you have a script on an enemy object that searchers for and fires at the player consider disabling this script when the enemy is too far for example from the player.

Examples

Fast and Efficient Checks

Avoid unnecessary operations and method calls wherever you can, especially in a method which is called many times a second, like `Update`.

Distance/Range Checks

Use `sqrMagnitude` instead of `magnitude` when comparing distances. This avoids unnecessary `sqrt` operations. Note that when using `sqrMagnitude`, the right hand side must also be squared.

```
if ((target.position - transform.position).sqrMagnitude < minDistance * minDistance))
```

Bounds Checks

Object intersections can be crudely checked by checking whether their `Collider`/`Renderer` bounds intersect. The `Bounds` structure also has a handy `Intersects` method which helps determine whether two bounds intersect.

`Bounds` also help us to calculate a close approximate of the *actual* (surface to surface) distance between objects (see `Bounds.SqrDistance`).

Caveats

Bounds checking works really well for convex objects, but bounds checks on concave objects may lead to much higher inaccuracies depending on the shape of the object.

Using `Mesh.bounds` is not recommended as it returns local space bounds. Use `MeshRenderer.bounds` instead.

Coroutine Power

Usage

If you have a long running operation that relies on the not-thread-safe Unity API, use Coroutines to split it over multiple frames and keep your application responsive.

Coroutines also help performing expensive actions every nth frame instead of running that action each frame.

Splitting Long-running Routines Over Multiple Frames

Coroutines help distribute long running operations over multiple frames to help keep up the framerate of your application.

Routines that paint or generate terrain procedurally or generate noise are examples that may need the Coroutine treatment.

```
for (int y = 0; y < heightmap.Height; y++)
{
    for (int x = 0; x < heightmap.Width; x++)
    {
        // Generate pixel at (x, y)
        // Assign pixel at (x, y)

        // Process only 32768 pixels each frame
        if ((y * heightmap.Height + x) % 32 * 1024) == 0)
            yield return null; // Wait for next frame
    }
}
```

> The code above is an easy to understand example. In production code it is better to avoid the per-pixel check that checks when to `yield return` (maybe do it every 2-3 rows) and to pre-calculate `for` loop length in advance.

Performing Expensive Actions Less Frequently

Coroutines help you perform expensive actions less frequently, so that it isn't as big a performance hit as it would be if performed every frame.

Taking the following example directly from the Manual:

```
private void ProximityCheck()
{
    for (int i = 0; i < enemies.Length; i++)
```

```
    {
        if (Vector3.Distance(transform.position, enemies[i].transform.position) <
dangerDistance)
                return true;
    }
    return false;
}

private IEnumerator ProximityCheckCoroutine()
{
    while(true)
    {
        ProximityCheck();
        yield return new WaitForSeconds(.1f);
    }
}
```

Proximity tests can be optimized even further by using the CullingGroup API.

Common Pitfalls

A common mistake developers make is accessing results or side effects of coroutines *outside* the coroutine. Coroutines return control to the caller as soon as a `yield return` statement is encountered and the result or side effect may not be performed yet. To circumvent problems where you *have* to use the result/side effect outside the coroutine, check this answer.

Strings

One might argue that there are greater resource hogs in Unity than the humble string, but it is one of the easier aspects to fix early on.

String operations build garbage

Most string operations build tiny amounts of garbage, but if those operations are called several times over the course of a single update, it stacks up. Over time it will trigger the automatic Garbage Collection, which may result in a visible CPU spike.

Cache your string operations

Consider the following example.

```
string[] StringKeys = new string[] {
    "Key0",
    "Key1",
    "Key2"
};

void Update()
{
```

```
    for (var i = 0; i < 3; i++)
    {
        // Cached, no garbage generated
        Debug.Log(StringKeys[i]);
    }

    for (var i = 0; i < 3; i++)
    {
        // Not cached, garbage every cycle
        Debug.Log("Key" + i);
    }

    // The most memory-efficient way is to not create a cache at all and use literals or
constants.
    // However, it is not necessarily the most readable or beautiful way.
    Debug.Log("Key0");
    Debug.Log("Key1");
    Debug.Log("Key2");
}
```

It may look silly and redundant, but if you're working with Shaders, you might run into situations such as these. Caching the keys will make a difference.

Please note that string *literals* and *constants* do not generate any garbage, as they are injected statically into the program stack space. If you are *generating* strings at run-time and are *guaranteed* to be generating the **same** strings each time like the above example, caching will definitely help.

For other cases where the string generated is not the same each time, there is no other alternative to generating those strings. As such, the memory spike with manually generating strings each time is usually negligible, unless tens of thousands of strings are being generated at a time.

Most string operations are Debug messages

Doing string operations for Debug messages, ie. `Debug.Log("Object Name: " + obj.name)` is fine and cannot be avoided during development. It is, however, important to ensure that irrelevant debug messages do not end up in the released product.

One way is to use the Conditional attribute in your debug calls. This not only removes the method calls, but also all the string operations going into it.

```
using UnityEngine;
using System.Collections;

public class ConditionalDebugExample: MonoBehaviour
{
    IEnumerator Start()
    {
        while(true)
        {
            // This message will pop up in Editor but not in builds
            Log("Elapsed: " + Time.timeSinceLevelLoad);
            yield return new WaitForSeconds(1f);
        }
```

```
    }

    [System.Diagnostics.Conditional("UNITY_EDITOR")]
    void Log(string Message)
    {
        Debug.Log(Message);
    }
}
```

> This is a simplified example. You might want to invest some time designing a more fully fledged logging routine.

String comparison

This is a minor optimisation, but it's worth a mention. Comparing strings is slightly more involved than one might think. The system will try to take cultural differences into account by default. You can opt to use a simple binary comparison instead, which performs faster.

```
// Faster string comparison
if (strA.Equals(strB, System.StringComparison.Ordinal)) {...}
// Compared to
if (strA == strB) {...}

// Less overhead
if (!string.IsNullOrEmpty(strA)) {...}
// Compared to
if (strA == "") {...}

// Faster lookups
Dictionary<string, int> myDic = new Dictionary<string, int>(System.StringComparer.Ordinal);
// Compared to
Dictionary<string, int> myDictionary = new Dictionary<string, int>();
```

Cache references

Cache references to avoid the expensive calls especially in the update function. This can be done by caching these references on start if available or when available and checking for null/bool flat to avoid getting the reference again.

Examples:

Cache component references

change

```
void Update()
{
    var renderer = GetComponent<Renderer>();
    renderer.material.SetColor("_Color", Color.green);
}
```

to

```
private Renderer myRenderer;
void Start()
{
    myRenderer = GetComponent<Renderer>();
}

void Update()
{
    myRenderer.material.SetColor("_Color", Color.green);
}
```

Cache object references

change

```
void Update()
{
    var enemy = GameObject.Find("enemy");
    enemy.transform.LookAt(new Vector3(0,0,0));
}
```

to

```
private Transform enemy;

void Start()
{
    this.enemy = GameObject.Find("enemy").transform;
}

void Update()
{
    enemy.LookAt(new Vector3(0, 0, 0));
}
```

Additionally cache expensive calls like calls to Mathf where possible.

Avoid calling methods using strings

Avoid calling methods using strings that can accept methods. This approach will make use of reflection that can slow down your game especially when used in the update function.

Examples:

```
    //Avoid StartCoroutine with method name
    this.StartCoroutine("SampleCoroutine");

    //Instead use the method directly
    this.StartCoroutine(this.SampleCoroutine());

    //Avoid send message
    var enemy = GameObject.Find("enemy");
    enemy.SendMessage("Die");

    //Instead make direct call
```

```
    var enemy = GameObject.Find("enemy") as Enemy;
    enemy.Die();
```

Avoid empty unity methods

Avoid empty unity methods. Apart from being bad programming style, there is a very small overhead involved in runtime scripting. Over many instances, this can build up and affect performance.

```
void Update
{
}

void FixedUpdate
{
}
```

Read Optimization online: https://riptutorial.com/unity3d/topic/3433/optimization

Chapter 25: Physics

Examples

Rigidbodies

Overview

The Rigidbody component gives a GameObject a *physical presence* in the scene in that it is able to respond to forces. You could apply forces directly to the GameObject or allow it to react to external forces such as gravity or another Rigidbody hitting it.

Adding a Rigidbody component

You can add a Rigidbody by clicking **Component > Physics > Rigidbody**

Moving a Rigidbody object

It is recommended that if you apply a Rigidbody to a GameObject that you use forces or torque to move it rather than manipulating it's Transform. Use `AddForce()` or `AddTorque()` methods for this:

```
// Add a force to the order of myForce in the forward direction of the Transform.
GetComponent<Rigidbody>().AddForce(transform.forward * myForce);

// Add torque about the Y axis to the order of myTurn.
GetComponent<Rigidbody>().AddTorque(transform.up * torque * myTurn);
```

Mass

You can alter the mass of a Rigidbody GameObject to affect how it reacts with other Rigidbodies and forces. A higher mass means the GameObject will have more of an influence on other physics-based GameObjects, and will require a greater force to move itself. Objects of differing mass will fall at the same rate if they have the same drag values. To alter mass in code:

```
GetComponent<Rigidbody>().mass = 1000;
```

Drag

The higher the drag value, the more an object will slow down while moving. Think of it like an opposing force. To alter drag in code:

```
GetComponent<Rigidbody>().drag = 10;
```

---

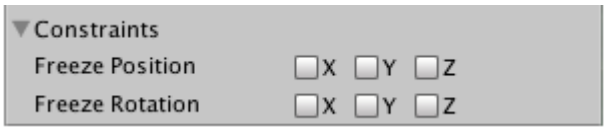
isKinematic

If you mark a Rigidbody as **Kinematic** then it cannot be affected by other forces but can still affect other GameObjects. To alter in code:

```
GetComponent<Rigidbody>().isKinematic = true;
```

Constraints

It is also possible to add constraints to each axis to freeze the Rigidbody's position or rotation in local space. The default is `RigidbodyConstraints.None` as shown here:



An example of constraints in code:

```
// Freeze rotation on all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeRotation

// Freeze position on all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePosition

// Freeze rotation and motion an all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeAll
```

You can use the bitwise OR operator `|` to combine multiple constraints like so:

```
// Allow rotation on X and Y axes and motion on Y and Z axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePositionZ |
    RigidbodyConstraints.FreezeRotationX;
```

Collisions

If you want a GameObject with a Rigidbody on it to respond to collisions you will also need to add a collider to it. Types of collider are:

- Box collider
- Sphere collider
- Capsule collider
- Wheel collider
- Mesh collider

If you apply more than one collider to a GameObject, we call it a Compound collider.

You can make a collider into a **Trigger** in order to use the `OnTriggerEnter()`, `OnTriggerStay()` and `OnTriggerExit()` methods. A trigger collider will not physically react to collisions, other GameObjects simply pass through it. They are useful for detecting when another GameObject is in a certain area or not, for example, when collecting an item, we may want to be able to just run through it but detect when this happens.

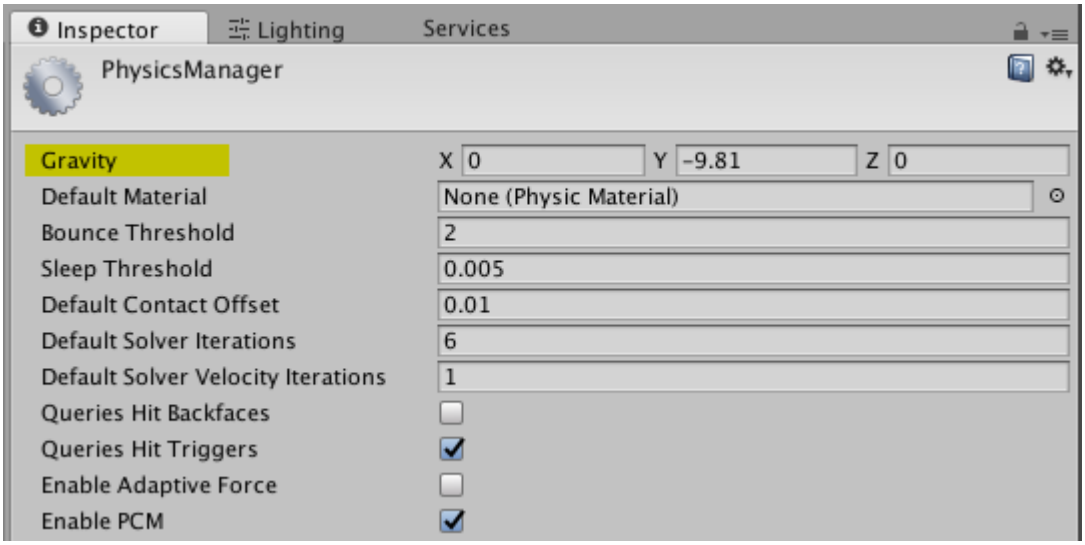
Gravity in Rigid Body

The `useGravity` property of a `RigidBody` controls whether gravity affects it or not. If set to `false` the `RigidBody` will behave as if in outer space (without a constant force being applied to it in some direction).

```
GetComponent<RigidBody>().useGravity = false;
```

It is very useful in the situations where you need all other properties of `RigidBody` except the motion controlled by gravity.

When enabled, the `RigidBody` will be affected by a gravitational force, set up under `Physics Settings` :



Gravity is defined in world units per second squared, and is entered here as a three-dimensional vector: meaning that with the settings in the example image, all `RigidBodies` with the `useGravity` property set to `True` will experience a force of 9.81 world units per second *per second* in the

downwards direction (as negative Y-values in Unity's coordinate system point downwards).

Read Physics online: https://riptutorial.com/unity3d/topic/3680/physics

Chapter 26: Prefabs

Syntax

- public static Object PrefabUtility.InstantiatePrefab(Object target);
- public static Object AssetDatabase.LoadAssetAtPath(string assetPath, Type type);
- public static Object Object.Instantiate(Object original);
- public static Object Resources.Load(string path);

Examples

Introduction

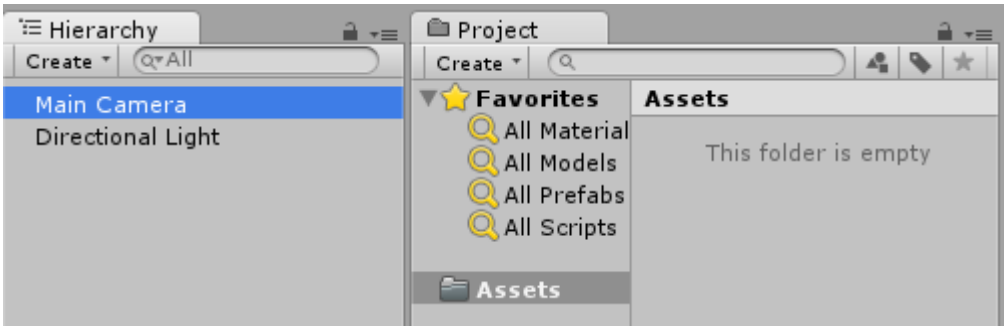
Prefabs are an asset type that allows the storage of a complete GameObject with its components, properties, attached components and serialized property values. There are many scenarios where this is useful, including:

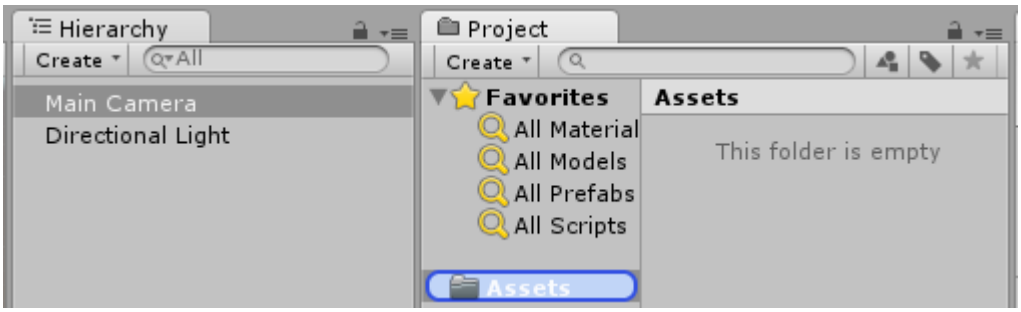
- Duplicating objects in a scene
- Sharing a common object across multiple scenes
- Being able to modify a prefab once and having the changes apply across multiple objects/scenes
- Creating duplicate objects with minor modifications, while having the common elements be editable from one base prefab
- Instantiating GameObjects at runtime

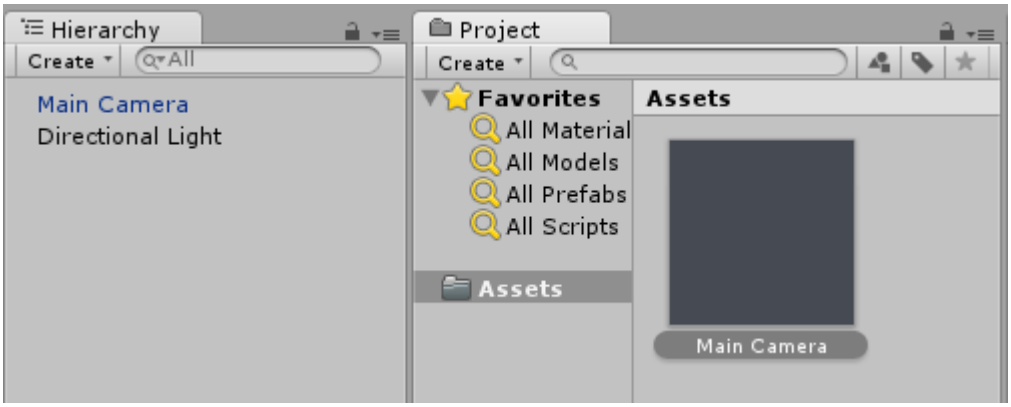
There is a rule of thumb of sorts in Unity that says "everything should be Prefabs". While this is probably exaggeration, it does encourage code reuse and the building of GameObjects in a reusable way, which is both memory efficient and good design.

Creating prefabs

To create a prefab, drag a game object from the scene hierarchy into the **Assets** folder or subfolder:





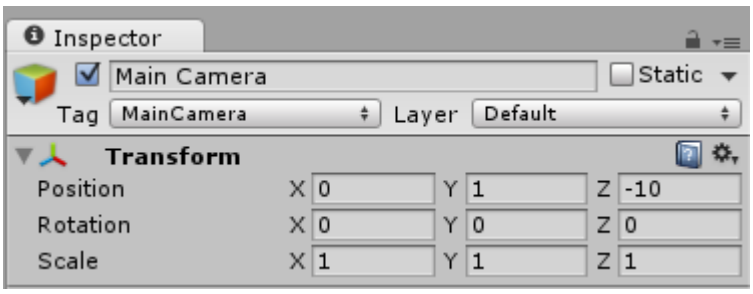


The game object name turns blue, indicating it is **connected to a prefab**.
Now this object is a **prefab instance**, just like an object instance of a class.

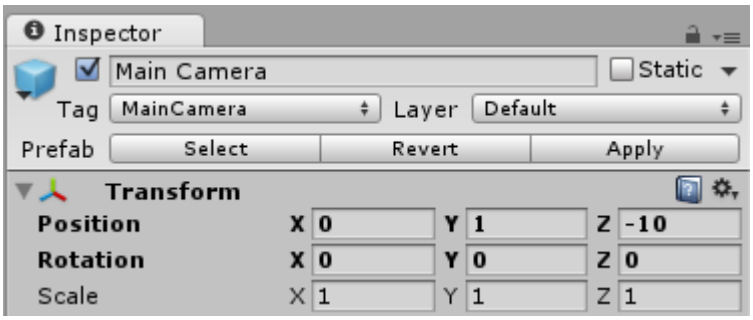
A prefab can be deleted after instantiation. In that case the name of the game object previously connected to it turns red:

Prefab inspector

If you select a prefab in the hierarchy view you'll notice that it's inspector slightly differs from an ordinary game object:



vs



Bold properties mean that their values differ from the prefab values. You can change any property of an instantiated prefab without affecting original prefab values. When a value is changed in a prefab instance, it turns bold and any subsequent changes of the same value in the prefab will not be reflected in the changed instance.

You can revert to original prefab values by clicking **Revert** button, which will also have value changes reflect in the instance. Additionally, to revert an individual value, you can right click it and press **Revert Value to Prefab**. To revert a component, right click it and press **Revert to Prefab**.

Clicking the **Apply** button overwrites prefab property values with the current game object property values. There is no "Undo" button or confirm dialog, so handle this button with care.

Select button highlights connected prefab in project's folder structure.

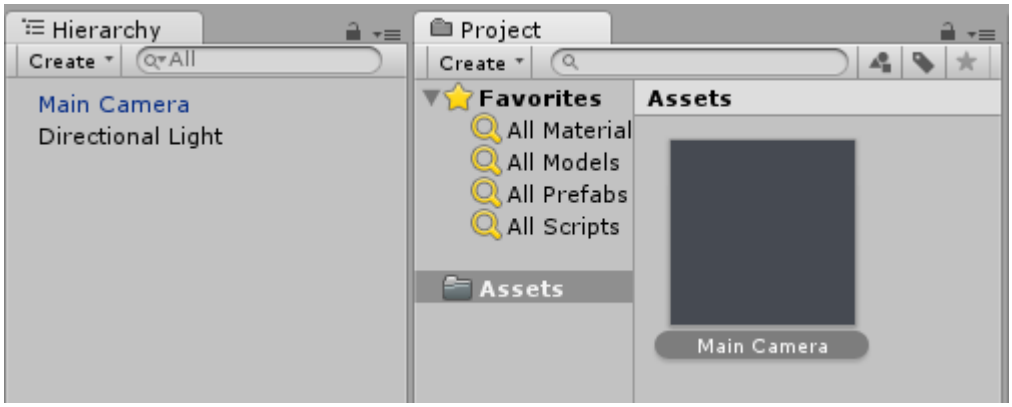
Instantiating prefabs

There are 2 ways of instantiating prefabs: during **design time** or **runtime**.

Design time instantiation

Instantiating prefabs at design time is useful to visually place multiple instances of the same object (e.g. *placing trees when designing a level of your game*).

- To visually instantiate a prefab drag it from the project view to scene hierarchy.



- If you are writing an editor extension, you can also instantiate a prefab programmatically calling `PrefabUtility.InstantiatePrefab()` method:

```
GameObject gameObject =
(GameObject)PrefabUtility.InstantiatePrefab(AssetDatabase.LoadAssetAtPath("Assets/MainCamera.pref
typeof(GameObject)));
```

Runtime instantiation

Instantiating prefabs at runtime is useful to create instances of an object according to some logic (e.g. *spawning an enemy every 5 seconds*).

To instantiate a prefab you need a reference to the prefab object. This can be done by having a `public GameObject` field in your `MonoBehaviour` script (and setting its value using the inspector in the Unity Editor):

```
public class SomeScript : MonoBehaviour {
   public GameObject prefab;
}
```

Or by putting the prefab in the Resource folder and using `Resources.Load`:

```
GameObject prefab = Resources.Load("Assets/Resources/MainCamera");
```

Once you have a reference to the prefab object you can instantiate it using the `Instantiate` function anywhere in your code (e.g. *inside a loop to create multiple objects*):

```
GameObject gameObject = Instantiate<GameObject>(prefab, new Vector3(0,0,0),
Quaternion.identity);
```

> Note: *Prefab* term does not exist at runtime.

Nested prefabs

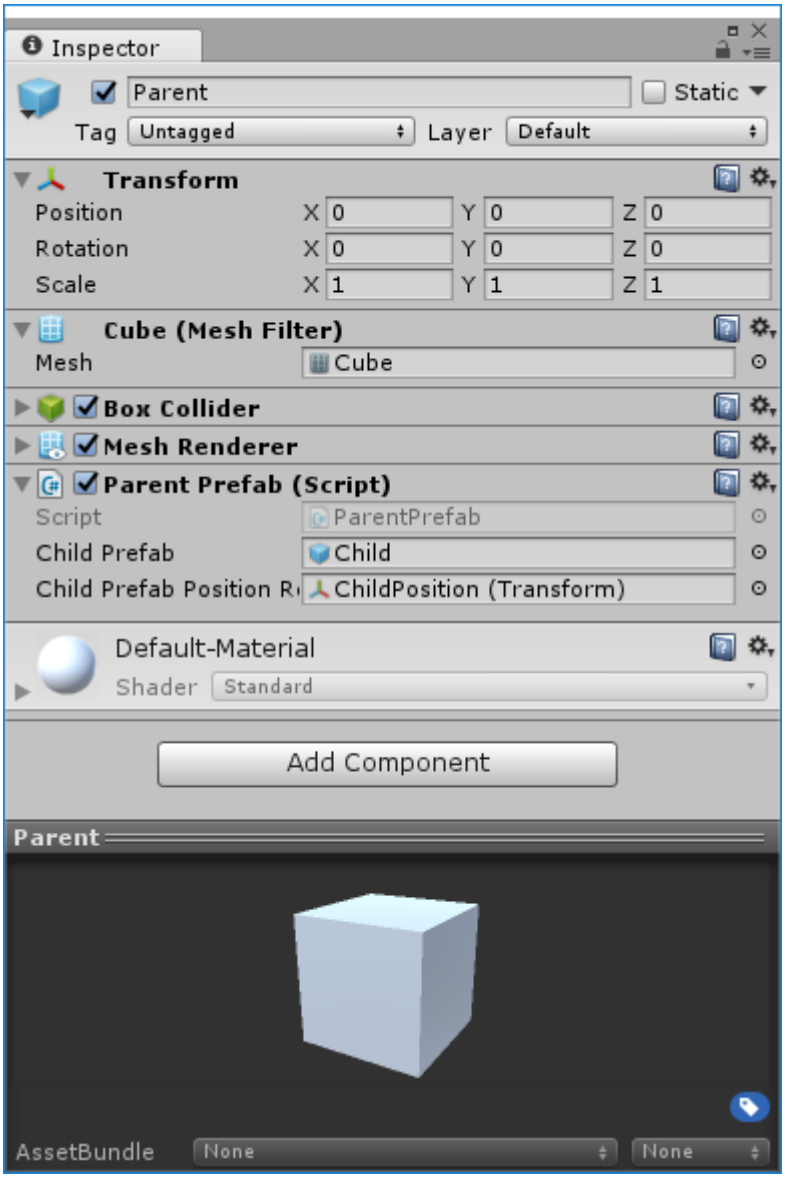
Nested prefabs are not avaible in Unity at the moment. You can drag one prefab to another, and apply that, but any changes on the child prefab will not be applied to nested one.

But there is a simple workaround - **You have to add to parent prefab a simple script, that will instantiate a child one.**

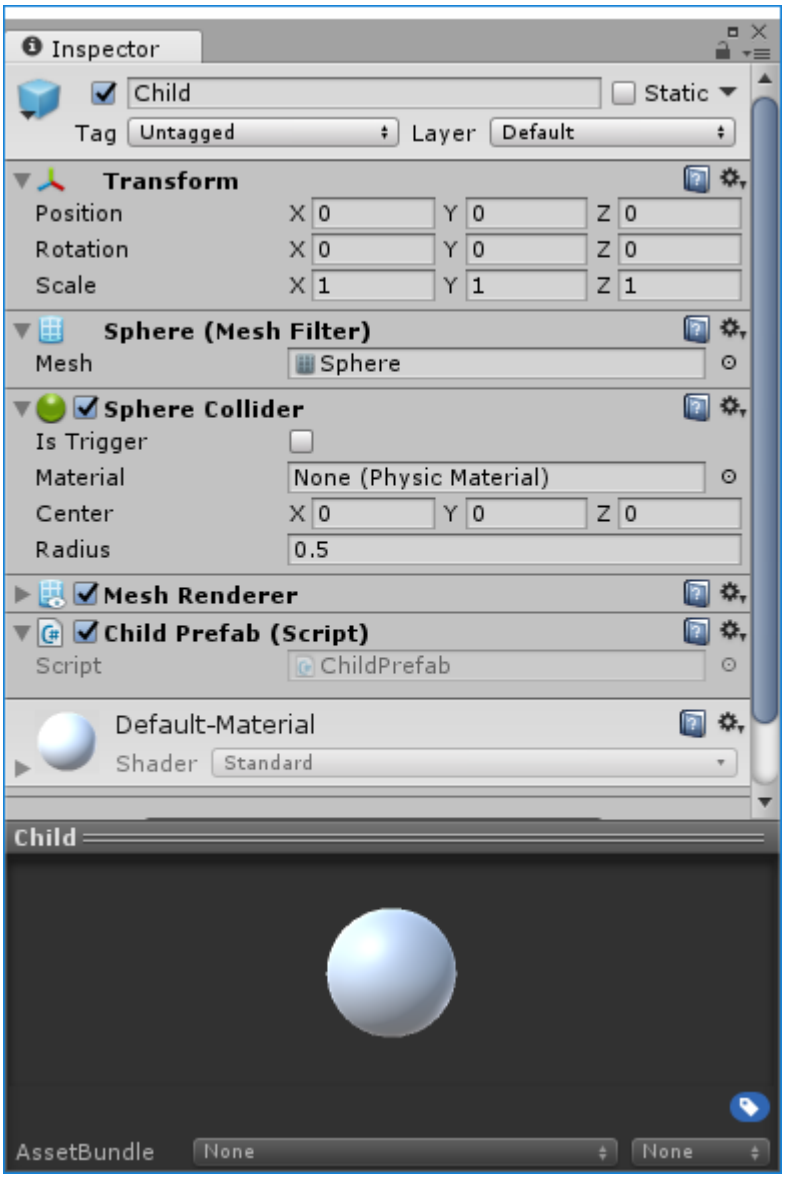
```
using UnityEngine;

public class ParentPrefab : MonoBehaviour {

    [SerializeField] GameObject childPrefab;
    [SerializeField] Transform childPrefabPositionReference;

    // Use this for initialization
    void Start () {
        print("Hello, I'm a parent prefab!");
        Instantiate(
            childPrefab,
```

```
            childPrefabPositionReference.position,
            childPrefabPositionReference.rotation,
            gameObject.transform
        );
    }
}
```

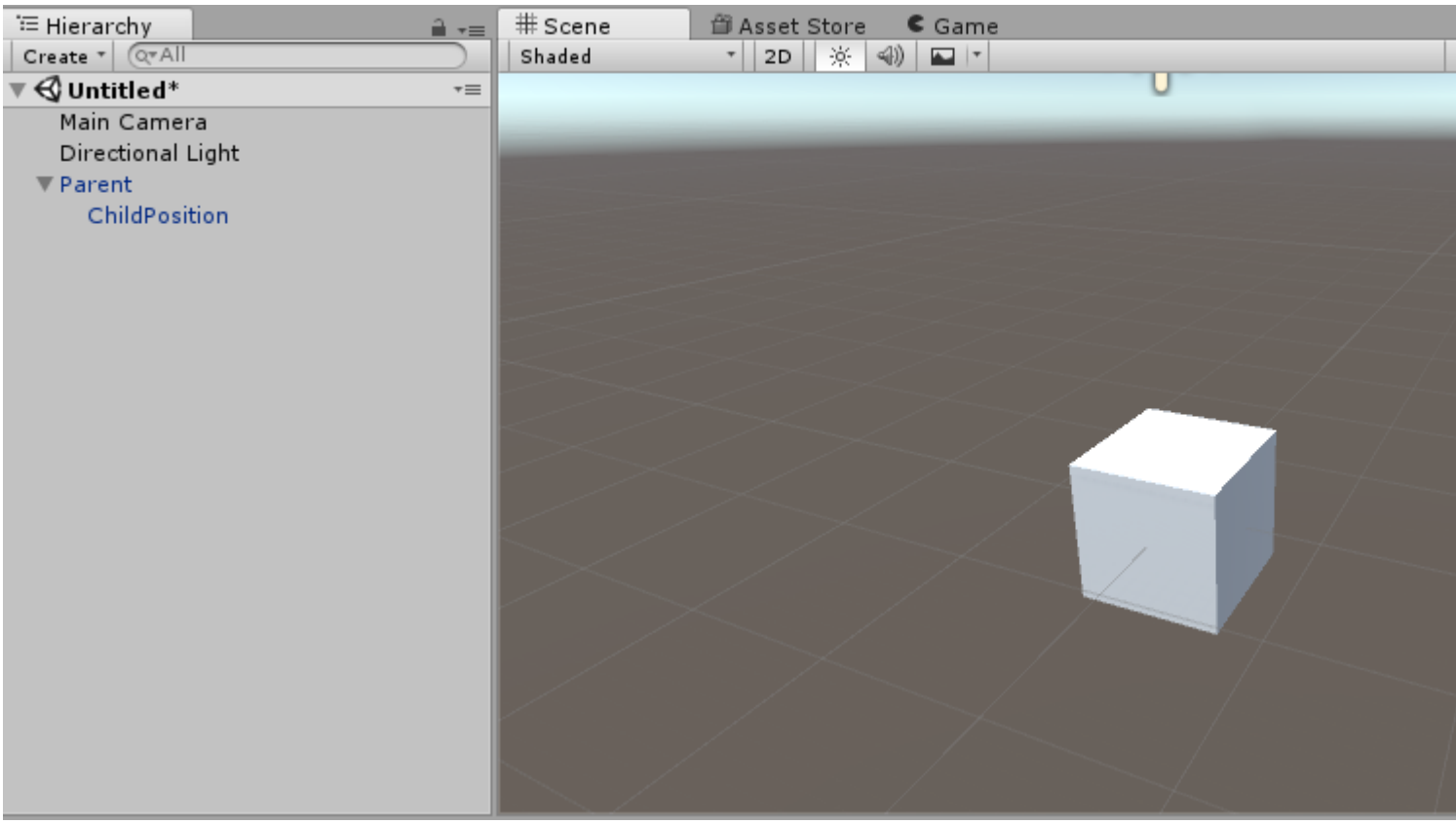
Parent prefab:

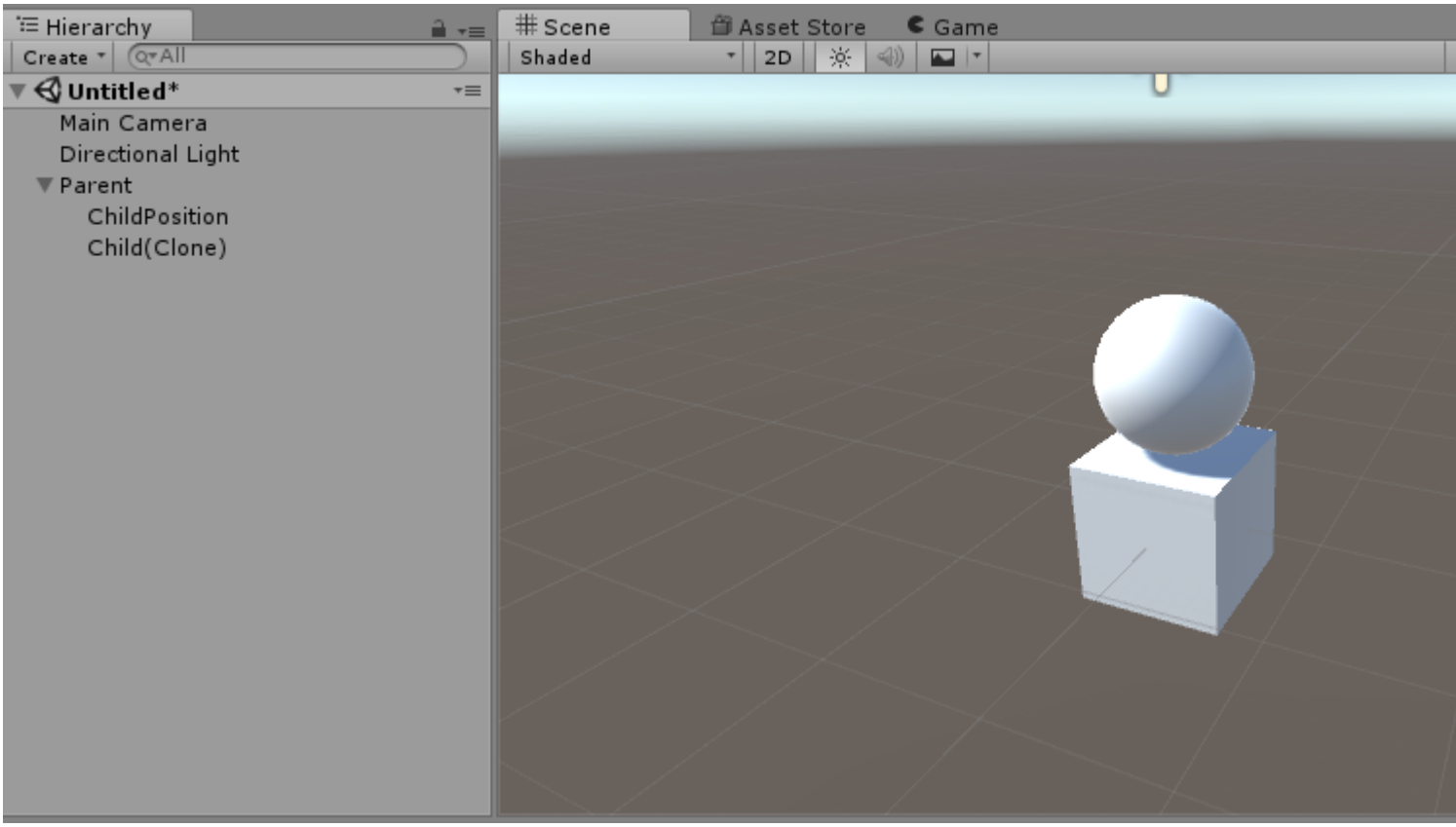


Child prefab:



Scene before and after start:





Read Prefabs online: https://riptutorial.com/unity3d/topic/2133/prefabs

Chapter 27: Quaternions

Syntax

- Quaternion.LookRotation(Vector3 forward [, Vector3 up]);
- Quaternion.AngleAxis(float angles, Vector3 axisOfRotation);
- float angleBetween = Quaternion.Angle(Quaternion rotation1, Quaternion rotation2);

Examples

Intro to Quaternion vs Euler

Euler angles are "degree angles" like 90, 180, 45, 30 degrees. Quaternions differ from Euler angles in that they represent a point on a Unit Sphere (the radius is 1 unit). You can think of this sphere as a 3D version of the Unit circle you learn in trigonometry. Quaternions differ from Euler angles in that they use imaginary numbers to define a 3D rotation.

While this may sound complicated (and arguably it is), Unity has great builtin functions that allow you to switch between Euler angles and quaterions, as well as functions to modify quaternions, without knowing a single thing about the math behind them.

Converting Between Euler and Quaternion

```
// Create a quaternion that represents 30 degrees about X, 10 degrees about Y
Quaternion rotation = Quaternion.Euler(30, 10, 0);

// Using a Vector
Vector3 EulerRotation = new Vector3(30, 10, 0);
Quaternion rotation = Quaternion.Euler(EulerRotation);

// Convert a transfroms Quaternion angles to Euler angles
Quaternion quaternionAngles = transform.rotation;
Vector3 eulerAngles = quaternionAngles.eulerAngles;
```

Why Use a Quaternion?

Quaternions solve a problem known as gimbal locking. This occurs when the primary axis of rotation becomes collinear with the tertiary axis of rotation. Here's a visual example @ 2:09

Quaternion Look Rotation

`Quaternion.LookRotation(Vector3 forward [, Vector3 up])` will create a Quaternion rotation that looks forward 'down' the forward vector and has the Y axis aligned with the 'up' vector. If the up vector is not specified, Vector3.up will be used.

Rotate this Game Object to look at a target Game Object

```
// Find a game object in the scene named Target
public Transform target = GameObject.Find("Target").GetComponent<Transform>();

// We subtract our position from the target position to create a
// Vector that points from our position to the target position
// If we reverse the order, our rotation would be 180 degrees off.
Vector3 lookVector = target.position - transform.position;
Quaternion rotation = Quaternion.LookRotation(lookVector);
transform.rotation = rotation;
```

Read Quaternions online: https://riptutorial.com/unity3d/topic/1782/quaternions

Chapter 28: Raycast

Parameters

Parameter	Details
origin	The starting point of the ray in world coordinates
direction	The direction of the ray
maxDistance	The max distance the ray should check for collisions
layerMask	A Layer mask that is used to selectively ignore Colliders when casting a ray.
queryTriggerInteraction	Specifies where this query should hit Triggers.

Examples

Physics Raycast

This function casts a ray from point `origin` in direction `direction` of length `maxDistance` against all colliders in the scene.

The function takes in the `origin direction maxDistance` and calculate if there is a collider in front of the GameObject.

```
Physics.Raycast(origin, direction, maxDistance);
```

For example, this function will print `Hello World` to the console if there is something within 10 units of the `GameObject` attached to it:

```
using UnityEngine;

public class TestPhysicsRaycast: MonoBehaviour
{
    void FixedUpdate()
    {
        Vector3 fwd = transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast(transform.position, fwd, 10))
            print("Hello World");
    }
}
```

Physics2D Raycast2D

You can use raycasts to check if an ai can walk without falling off the edge of a level.

```
using UnityEngine;

public class Physics2dRaycast: MonoBehaviour
    {
        public LayerMask LineOfSightMask;
        void FixedUpdate()
        {
            RaycastHit2D hit = Physics2D.Raycast(raycastRightPart, Vector2.down, 0.6f *
heightCharacter, LineOfSightMask);
            if(hit.collider != null)
            {
                //code when the ai can walk
            }
            else
            {
                //code when the ai cannot walk
            }
    }
}
```

In this example the direction is right. The variable raycastRightPart is the right part of the character, so the raycast will happen at the right part of the character. The distance is 0.6f times the height of the character so the raycast won't give a hit when he hits the ground that is way lower than the ground he is standing on at the moment. Make sure the Layermask is set to ground only, otherwise it will detect other kinds of objects as well.

> RaycastHit2D itself is a structure and not a class so hit can't be null; this means you have to check for the collider of a RaycastHit2D variable.

Encapsulating Raycast calls

Having your scripts call `Raycast` directly may lead to problems if you need to change the collision matrices in the future, as you'll have to track down every `LayerMask` field to accommodate the changes. Depending on the size of your project, this may become a huge undertaking.

Encapsulating `Raycast` calls may make your life easier down the line.

Looking at it from a SoC principle, a gameobject really shouldn't know or care about LayerMasks. It only needs a method to scan its surroundings. Whether the raycast result returns this or that shouldn't matter to the gameobject. It should only act upon the information it receives and not make any assumptions on the environment it exists in.

One way to approach this is to move the LayerMask value to ScriptableObject instances and use those as a form of raycast services that you inject into your scripts.

```
// RaycastService.cs
using UnityEngine;

[CreateAssetMenu(menuName = "StackOverflow")]
public class RaycastService : ScriptableObject
{
```

```
    [SerializeField]
    LayerMask layerMask;

    public RaycastHit2D Raycast2D(Vector2 origin, Vector2 direction, float distance)
    {
        return Physics2D.Raycast(origin, direction, distance, layerMask.value);
    }

    // Add more methods as needed

}
```

```
// MyScript.cs
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [SerializeField]
    RaycastService raycastService;

    void FixedUpdate()
    {
        RaycastHit2D hit = raycastService.Raycast2D(Vector2.zero, Vector2.down, 1f);
    }
}
```

This allows you to make a number of raycast services, all with different LayerMask combinations for different situations. You could have one that hits only ground colliders, and another that hits ground colliders and one way platforms.

If you ever need to make drastic changes to your LayerMask setups, you only need to update these RaycastService assets.

Further reading

- Inversion of control
- Dependency injection

Read Raycast online: https://riptutorial.com/unity3d/topic/2826/raycast

Chapter 29: Resources

Examples

Introduction

With Resources class it's possible to dynamically load assets that are not part of the scene. It's very usefull when you have to use on demand assets, for example localize multi language audios, texts, etc..

Assets must be placed in a folder named **Resources**. It's possible to have multiple Resources folders spread across the project's hierarchy. `Resources` class will inspect all Resources folders you may have.

Every asset placed within Resources will be included in the build even it's not referenced in your code. Thus don't insert assets in Resources indiscriminately.

```
//Example of how to load language specific audio from Resources

[RequireComponent(typeof(AudioSource))]
public class loadIntroAudio : MonoBehaviour {
    void Start () {
        string language = Application.systemLanguage.ToString();
        AudioClip ac = Resources.Load(language + "/intro") as AudioClip; //loading intro.mp3
specific for user's language (note the file file extension should not be used)
        if (ac==null)
        {
            ac = Resources.Load("English/intro") as AudioClip; //fallback to the english
version for any unsupported language
        }
        transform.GetComponent<AudioSource>().clip = ac;
        transform.GetComponent<AudioSource>().Play();
    }
}
```

Resources 101

Introduction

Unity has a few 'specially named' folders that allows for a variety of uses. One of these folders is called 'Resources'

The 'Resources' folder is one of only TWO ways of loading assets at runtime in Unity (The other being AssetBundles (Unity Docs)

The 'Resources' folder can reside anywhere inside your Assets folder, and you can have multiple folders named Resources. The contents of all 'Resources' folders are merged during compile time.

The primary way to load an asset from a Resources folder is to use the Resources.Load function. This function takes a string parameter which allows you to specify the path of the file **relative** to the Resources folder. Note that you do NOT need to specify file extensions while loading an asset

```
public class ResourcesSample : MonoBehaviour {

    void Start () {
        //The following line will load a TextAsset named 'foobar' which was previously place
under 'Assets/Resources/Stackoverflow/foobar.txt'
        //Note the absence of the '.txt' extension! This is important!

        var text = Resources.Load<TextAsset>("Stackoverflow/foobar").text;
        Debug.Log(string.Format("The text file had this in it :: {0}", text));
    }
}
```

Objects which are comprised of multiple objects can also be loaded from Resources. Examples are such objects are 3D models with textures baked in, or a multiple sprite.

```
//This example will load a multiple sprite texture from Resources named "A_Multiple_Sprite"
var sprites = Resources.LoadAll("A_Multiple_Sprite") as Sprite[];
```

Putting it all together

Here's one of my helper classes which I use to load all sounds for any game. You can attach this to any GameObject in a scene and it will load the specified audio files from the 'Resources/Sounds' folder

```
public class SoundManager : MonoBehaviour {

    void Start () {

        //An array of all sounds you want to load
        var filesToLoad = new string[] { "Foo", "Bar" };

        //Loop over the array, attach an Audio source for each sound clip and assign the
        //clip property.
        foreach(var file in filesToLoad) {
            var soundClip = Resources.Load<AudioClip>("Sounds/" + file);
            var audioSource = gameObject.AddComponent<AudioSource>();
            audioSource.clip = soundClip;
        }
    }
}
```

Final Notes

1. Unity is smart when it comes to including assets into your build. Any asset that is not serialized (i.e. used in a scene that's included in a build) is excluded from a build. HOWEVER this DOES NOT apply to any asset inside the Resources folder. Therefore, do not go overboard on adding assets to this folder

2. Assets that are loaded using Resources.Load or Resources.LoadAll can be unloaded in the future by using Resources.UnloadUnusedAssets or Resources.UnloadAsset

Read Resources online: https://riptutorial.com/unity3d/topic/4070/resources

Chapter 30: ScriptableObject

Remarks

ScriptableObjects with AssetBundles

Pay attention when adding prefabs to AssetBundles if they contain references to ScriptableObjects. Since ScriptableObjects are essentially assets, Unity creates duplicates of them before adding them to AssetBundles, which may result in undesired behaviour during runtime.

When you load such a GameObject from an AssetBundle, it may be necessary to reinject the ScriptableObject assets to the loaded scripts, replacing the bundled ones. See Dependency Injection

Examples

Introduction

ScriptableObjects are serialized objects that are not bound to scenes or gameobjects as MonoBehaviours are. To put it one way, they are data and methods bound to asset files inside your project. These ScriptableObject assets can be passed to MonoBehaviours or other ScriptableObjects, where their public methods can be accessed.

Due to their nature as serialized assets, they make for excellent manager classes and data sources.

Creating ScriptableObject assets

Below is a simple ScriptableObject implementation.

```
using UnityEngine;

[CreateAssetMenu(menuName = "StackOverflow/Examples/MyScriptableObject")]
public class MyScriptableObject : ScriptableObject
{
    [SerializeField]
    int mySerializedNumber;

    int helloWorldCount = 0;

    public void HelloWorld()
    {
        helloWorldCount++;
        Debug.LogFormat("Hello! My number is {0}.", mySerializedNumber);
        Debug.LogFormat("I have been called {0} times.", helloWorldCount);
```

```
    }
}
```

By adding the `CreateAssetMenu` attribute to the class, Unity will list it in the **Assets/Create** submenu. In this case it's under **Assets/Create/StackOverflow/Examples**.

Once created, ScriptableObject instances can be passed to other scripts and ScriptableObjects through the Inspector.

```
using UnityEngine;

public class SampleScript : MonoBehaviour {

    [SerializeField]
    MyScriptableObject myScriptableObject;

    void OnEnable()
    {
        myScriptableObject.HelloWorld();
    }
}
```

Create ScriptableObject instances through code

You create new ScriptableObject instances through `ScriptableObject.CreateInstance<T>()`

```
T obj = ScriptableObject.CreateInstance<T>();
```

Where `T` extends `ScriptableObject`.

> Do not create ScriptableObjects by calling their constructors, ie. `new ScriptableObject()`.

Creating ScriptableObjects by code during runtime is rarely called for because their main use is data serialization. You might as well use standard classes at this point. It is more common when you are scripting editor extensions.

ScriptableObjects are serialized in editor even in PlayMode

Extra care should be taken when accessing serialized fields in a ScriptableObject instance.

If a field is marked `public` or serialized through `SerializeField`, changing its value is permanent. They do not reset when exiting playmode like MonoBehaviours do. This can be useful at times, but it can also make a mess.

Because of this it's best to make serialized fields read-only and avoid public fields altogether.

```
public class MyScriptableObject : ScriptableObject
{
    [SerializeField]
    int mySerializedValue;

    public int MySerializedValue
```

```
    {
        get { return mySerializedValue; }
    }
}
```

If you wish to store public values in a ScriptableObject that are reset between play sessions, consider using the following pattern.

```
public class MyScriptableObject : ScriptableObject
{
    // Private fields are not serialized and will reset to default on reset
    private int mySerializedValue;

    public int MySerializedValue
    {
        get { return mySerializedValue; }
        set { mySerializedValue = value; }
    }
}
```

Find existing ScriptableObjects during runtime

To find *active* ScriptableObjects during runtime, you can use `Resources.FindObjectsOfTypeAll()`.

```
T[] instances = Resources.FindObjectsOfTypeAll<T>();
```

Where `T` is the type of the ScriptableObject instance you're searching. *Active* means it has been loaded in memory in some form before.

This method is very slow so remember to cache the return value and avoid calling it frequently. Referencing the ScriptableObjects directly in your scripts should be your preferred option.

> *Tip:* You can maintain your own instance collections for faster lookups. Have your ScriptableObjects register themselves to a shared collection during `OnEnable()`.

Read ScriptableObject online: https://riptutorial.com/unity3d/topic/3434/scriptableobject

Chapter 31: Singletons in Unity

Remarks

While there are schools of thought which make compelling arguments why unconstrained use of Singletons is a bad idea, e.g. Singleton on gameprogrammingpatterns.com, there are occasions when you might want to persist a GameObject in Unity over multiple Scenes (e.g. for seamless background music) while ensuring that no more than one instance can exist; a perfect use case for a Singleton.

By adding this script to a GameObject, once it has been instantiated (e.g. by including it anywhere in a Scene) it will remain active across Scenes, and only one instance will ever exist.

ScriptableObject (UnityDoc) instances provide a valid alternative to Singletons for some use cases. While they don't implicitly enforce the single instance rule, they retain their state between scenes and play nicely with the Unity serialization process. They also promote Inversion of Control as dependencies are injected through the editor.

```
// MyAudioManager.cs
using UnityEngine;

[CreateAssetMenu] // Remember to create the instance in editor
public class MyAudioManager : ScriptableObject {
    public void PlaySound() {}
}
```

```
// MyGameObject.cs
using UnityEngine;

public class MyGameObject : MonoBehaviour
{
    [SerializeField]
    MyAudioManager audioManager; //Insert through Inspector

    void OnEnable()
    {
        audioManager.PlaySound();
    }
}
```

Further reading

- Singleton Implementation in C#

Examples

Implementation using RuntimeInitializeOnLoadMethodAttribute

Since **Unity 5.2.5** it's possible to use RuntimeInitializeOnLoadMethodAttribute to execute initialization logic bypassing MonoBehaviour order of execution. It provides a way to create more clean and robust implementation:

```
using UnityEngine;

sealed class GameDirector : MonoBehaviour
{
    // Because of using RuntimeInitializeOnLoadMethod attribute to find/create and
    // initialize the instance, this property is accessible and
    // usable even in Awake() methods.
    public static GameDirector Instance
    {
        get; private set;
    }

    // Thanks to the attribute, this method is executed before any other MonoBehaviour
    // logic in the game.
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    static void OnRuntimeMethodLoad()
    {
        var instance = FindObjectOfType<GameDirector>();

        if (instance == null)
            instance = new GameObject("Game Director").AddComponent<GameDirector>();

        DontDestroyOnLoad(instance);

        Instance = instance;
    }

    // This Awake() will be called immediately after AddComponent() execution
    // in the OnRuntimeMethodLoad(). In other words, before any other MonoBehaviour's
    // in the scene will begin to initialize.
    private void Awake()
    {
        // Initialize non-MonoBehaviour logic, etc.
        Debug.Log("GameDirector.Awake()", this);
    }
}
```

The resulting order of execution:

1. `GameDirector.OnRuntimeMethodLoad()` started...
2. `GameDirector.Awake()`
3. `GameDirector.OnRuntimeMethodLoad()` completed.
4. `OtherMonoBehaviour1.Awake()`
5. `OtherMonoBehaviour2.Awake()`, etc.

A simple Singleton MonoBehaviour in Unity C#

In this example, a private static instance of the class is declared at its beginning.

The value of a static field is shared between instances, so if a new instance of this class gets

created the `if` will find a reference to the first Singleton object, destroying the new instance (or its game object).

```
using UnityEngine;

public class SingletonExample : MonoBehaviour {

    private static SingletonExample _instance;

    void Awake(){

        if (_instance == null){

            _instance = this;
            DontDestroyOnLoad(this.gameObject);

            //Rest of your Awake code

        } else {
            Destroy(this);
        }
    }

    //Rest of your class code

}
```

Advanced Unity Singleton

This example combines multiple variants of MonoBehaviour singletons found on the internet into one and let you change its behavior depending on global static fields.

This example was tested using Unity 5. To use this singleton, all you need to do is extend it as follows: `public class MySingleton : Singleton<MySingleton> {}`. You may also need to override `AwakeSingleton` to use it instead of usual `Awake`. For further tweaking, change default values of static fields as described below.

---

1. This implementation makes use of DisallowMultipleComponent attribute to keep one instance per GameObject.
2. This class is abstract and can only be extended. It also contains one virtual method `AwakeSingleton` that needs to be overridden instead of implementing normal `Awake`.
3. This implementation is thread safe.
4. This singleton is optimized. By using `instantiated` flag instead of instance null check we avoid the overhead that comes with Unity's implementation of `==` operator. (Read more)
5. This implementation does not allow any calls to the singleton instance when it's about to get destroyed by Unity.
6. This singleton comes with the following options:

- `FindInactive`: whether to look for other instances of components of same type attached to inactive GameObject.
- `Persist`: whether to keep component alive between scenes.

- `DestroyOthers`: whether to destroy any other components of same type and keep only one.
- `Lazy`: whether to set singleton instance "on the fly" (in `Awake`) or only "on demand" (when getter is called).

```
using UnityEngine;

[DisallowMultipleComponent]
public abstract class Singleton<T> : MonoBehaviour where T : Singleton<T>
{
    private static volatile T instance;
    // thread safety
    private static object _lock = new object();
    public static bool FindInactive = true;
    // Whether or not this object should persist when loading new scenes. Should be set in
Init().
    public static bool Persist;
    // Whether or not destory other singleton instances if any. Should be set in Init().
    public static bool DestroyOthers = true;
    // instead of heavy comparision (instance != null)
    // http://blogs.unity3d.com/2014/05/16/custom-operator-should-we-keep-it/
    private static bool instantiated;

    private static bool applicationIsQuitting;

    public static bool Lazy;

    public static T Instance
    {
        get
        {
            if (applicationIsQuitting)
            {
                Debug.LogWarningFormat("[Singleton] Instance '{0}' already destroyed on
application quit. Won't create again - returning null.", typeof(T));
                return null;
            }
            lock (_lock)
            {
                if (!instantiated)
                {
                    Object[] objects;
                    if (FindInactive) { objects = Resources.FindObjectsOfTypeAll(typeof(T)); }
                    else { objects = FindObjectsOfType(typeof(T)); }
                    if (objects == null || objects.Length < 1)
                    {
                        GameObject singleton = new GameObject();
                        singleton.name = string.Format("{0} [Singleton]", typeof(T));
                        Instance = singleton.AddComponent<T>();
                        Debug.LogWarningFormat("[Singleton] An Instance of '{0}' is needed in
the scene, so '{1}' was created{2}", typeof(T), singleton.name, Persist ? " with
DontDestoryOnLoad." : ".");
                    }
                    else if (objects.Length >= 1)
                    {
                        Instance = objects[0] as T;
                        if (objects.Length > 1)
                        {
                            Debug.LogWarningFormat("[Singleton] {0} instances of '{1}'!",
objects.Length, typeof(T));
                            if (DestroyOthers)
```

```
                            {
                                for (int i = 1; i < objects.Length; i++)
                                {
                                    Debug.LogWarningFormat("[Singleton] Deleting extra '{0}'
instance attached to '{1}'", typeof(T), objects[i].name);
                                    Destroy(objects[i]);
                                }
                            }
                        }
                        return instance;
                    }
                }
                return instance;
            }
        }
        protected set
        {
            instance = value;
            instantiated = true;
            instance.AwakeSingleton();
            if (Persist) { DontDestroyOnLoad(instance.gameObject); }
        }
    }

    // if Lazy = false and gameObject is active this will set instance
    // unless instance was called by another Awake method
    private void Awake()
    {
        if (Lazy) { return; }
        lock (_lock)
        {
            if (!instantiated)
            {
                Instance = this as T;
            }
            else if (DestroyOthers && Instance.GetInstanceID() != GetInstanceID())
            {
                Debug.LogWarningFormat("[Singleton] Deleting extra '{0}' instance attached to
'{1}'", typeof(T), name);
                Destroy(this);
            }
        }
    }

    // this might be called for inactive singletons before Awake if FindInactive = true
    protected virtual void AwakeSingleton() {}

    protected virtual void OnDestroy()
    {
        applicationIsQuitting = true;
        instantiated = false;
    }
}
```

Singleton Implementation through base class

In projects that feature several singleton classes (as is often the case), it can be clean and convenient to abstract the singleton behaviour to a base class:

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

public abstract class MonoBehaviourSingleton<T> : MonoBehaviour {

    private static Dictionary<Type, object> _singletons
        = new Dictionary<Type, object>();

    public static T Instance {
        get {
            return (T)_singletons[typeof(T)];
        }
    }

    void OnEnable() {
        if (_singletons.ContainsKey(GetType())) {
            Destroy(this);
        } else {
            _singletons.Add(GetType(), this);
            DontDestroyOnLoad(this);
        }
    }
}
```

A MonoBehaviour may then implement the singleton pattern by extending MonoBehaviourSingleton. This approach allows the pattern to be utilised with a minimal footprint on the Singleton itself:

```
using UnityEngine;
using System.Collections;

public class SingletonImplementation : MonoBehaviourSingleton<SingletonImplementation> {

    public string Text= "String Instance";

    // Use this for initialisation
    IEnumerator Start () {
        var demonstration = "SingletonImplementation.Start()\n" +
                            "Note that the this text logs only once and\n"
                            "only one class instance is allowed to exist.";
        Debug.Log(demonstration);
        yield return new WaitForSeconds(2f);
        var secondInstance = new GameObject();
        secondInstance.AddComponent<SingletonImplementation>();
    }

}
```

Note that one of the benefits of the singleton pattern is that a reference to the instance may be accessed statically:

```
// Logs: String Instance
Debug.Log(SingletonImplementation.Instance.Text);
```

Keep in mind though, this practise should be minimised in order to reduce coupling. This approach also comes at a slight performance cost due to the use of Dictionary, but as this collection may

contain only one instance of each singleton class, the trade-off in terms of the DRY principle (Don't Repeat Yourself), readability and convenience is small.

Singleton Pattern utilizing Unitys Entity-Component system

The core idea is to use GameObjects to represent singletons, which has multiple advantages:

- Keeps complexity to a minimum but supports concepts like dependency injection
- Singletons have a normal Unity lifecycle as part of the Entity-Component system
- Singletons can be lazy loaded and cached locally where regulary needed (e.g. in update loops)
- No static fields needed
- No need to modify existing MonoBehaviours / Components to use them as Singletons
- Easy to reset (just destroy the Singletons GameObject), will be lazy loaded again on next usage
- Easy to inject mocks (just initialize it with the mock before using it)
- Inspection and configuration using normal Unity editor and can happen already on editor time (Screenshot of a Singleton accessible in the Unity editor)

Test.cs (which uses the example singleton):

```
using UnityEngine;
using UnityEngine.Assertions;

public class Test : MonoBehaviour {
    void Start() {
        ExampleSingleton singleton = ExampleSingleton.instance;
        Assert.IsNotNull(singleton); // automatic initialization on first usage
        Assert.AreEqual("abc", singleton.myVar1);
        singleton.myVar1 = "123";
        // multiple calls to instance() return the same object:
        Assert.AreEqual(singleton, ExampleSingleton.instance);
        Assert.AreEqual("123", ExampleSingleton.instance.myVar1);
    }
}
```

ExampleSingleton.cs (which contains an example and the actual Singleton class):

```
using UnityEngine;
using UnityEngine.Assertions;

public class ExampleSingleton : MonoBehaviour {
    public static ExampleSingleton instance { get { return Singleton.get<ExampleSingleton>();
} }
    public string myVar1 = "abc";
    public void Start() { Assert.AreEqual(this, instance, "Singleton more than once in
scene"); }
}

/// <summary> Helper that turns any MonBehaviour or other Component into a Singleton
</summary>
public static class Singleton {
    public static T get<T>() where T : Component {
        return GetOrAddGo("Singletons").GetOrAddChild("" + typeof(T)).GetOrAddComponent<T>();
```

```
    }
    private static GameObject GetOrAddGo(string goName) {
        var go = GameObject.Find(goName);
        if (go == null) { return new GameObject(goName); }
        return go;
    }
}

public static class GameObjectExtensionMethods {
    public static GameObject GetOrAddChild(this GameObject parentGo, string childName) {
        var childGo = parentGo.transform.FindChild(childName);
        if (childGo != null) { return childGo.gameObject; } // child found, return it
        var newChild = new GameObject(childName);        // no child found, create it
        newChild.transform.SetParent(parentGo.transform, false); // add it to parent
        return newChild;
    }

    public static T GetOrAddComponent<T>(this GameObject parentGo) where T : Component {
        var comp = parentGo.GetComponent<T>();
        if (comp == null) { return parentGo.AddComponent<T>(); }
        return comp;
    }
}
```

The two extension methods for GameObject are helpful in other situations as well, if you don't need them move them inside the Singleton class and make them private.

MonoBehaviour & ScriptableObject based Singleton Class

Most Singleton examples use MonoBehaviour as the base class. The main disadvantage is that this Singleton class only lives during run time. This has some drawbacks:

- There is no way of directly editing the singleton fields other than changing the code.
- No way to store a reference to other assets on the Singleton.
- No way of setting the singleton as the destination of a Unity UI event. I end up using what i call "Proxy Components" that its sole propose is to have 1 line methods that call "GameManager.Instance.SomeGlobalMethod()".

As noted on the remarks there are implementations that try to solve this using ScriptableObjects as base class but lose the run time benefits of the MonoBehaviour. This implementation solves this problems by using a ScriptableObject as a base class and an associated MonoBehavior during run time:

- It is an asset so its properties can be updated on the editor like any other Unity asset.
- It plays nicely with the Unity serialization process.
- Is possible to assign references on the singleton to other assets from the editor (dependencies are injected through the editor).
- Unity events can directly call methods on the Singleton.
- Can call it from anywhere in the codebase using "SingletonClassName.Instance"
- Has access to run time MonoBehaviour events and methods like: Update, Awake, Start, FixedUpdate, StartCoroutine, etc.

```
/*********************************************************
 * Better Singleton by David Darias
 * Use as you like - credit where due would be appreciated :D
 * Licence: WTFPL V2, Dec 2014
 * Tested on Unity v5.6.0 (should work on earlier versions)
 * 03/02/2017 - v1.1
 * *******************************************************/

using System;
using UnityEngine;
using SingletonScriptableObjectNamespace;

public class SingletonScriptableObject<T> :
SingletonScriptableObjectNamespace.BehaviourScriptableObject where T :
SingletonScriptableObjectNamespace.BehaviourScriptableObject
{
    //Private reference to the scriptable object
    private static T _instance;
    private static bool _instantiated;
    public static T Instance
    {
        get
        {
            if (_instantiated) return _instance;
            var singletonName = typeof(T).Name;
            //Look for the singleton on the resources folder
            var assets = Resources.LoadAll<T>("");
            if (assets.Length > 1) Debug.LogError("Found multiple " + singletonName + "s on
the resources folder. It is a Singleton ScriptableObject, there should only be one.");
            if (assets.Length == 0)
            {
                _instance = CreateInstance<T>();
                Debug.LogError("Could not find a " + singletonName + " on the resources
folder. It was created at runtime, therefore it will not be visible on the assets folder and
it will not persist.");
            }
            else _instance = assets[0];
            _instantiated = true;
            //Create a new game object to use as proxy for all the MonoBehaviour methods
            var baseObject = new GameObject(singletonName);
            //Deactivate it before adding the proxy component. This avoids the execution of
the Awake method when the the proxy component is added.
            baseObject.SetActive(false);
            //Add the proxy, set the instance as the parent and move to DontDestroyOnLoad
scene
            SingletonScriptableObjectNamespace.BehaviourProxy proxy =
baseObject.AddComponent<SingletonScriptableObjectNamespace.BehaviourProxy>();
            proxy.Parent = _instance;
            Behaviour = proxy;
            DontDestroyOnLoad(Behaviour.gameObject);
            //Activate the proxy. This will trigger the MonoBehaviourAwake.
            proxy.gameObject.SetActive(true);
            return _instance;
        }
    }
    //Use this reference to call MonoBehaviour specific methods (for example StartCoroutine)
    protected static MonoBehaviour Behaviour;
    public static void BuildSingletonInstance() {
SingletonScriptableObjectNamespace.BehaviourScriptableObject i = Instance; }
    private void OnDestroy(){ _instantiated = false; }
}
```

```
// Helper classes for the SingletonScriptableObject
namespace SingletonScriptableObjectNamespace
{
    #if UNITY_EDITOR
    //Empty custom editor to have cleaner UI on the editor.
    using UnityEditor;
    [CustomEditor(typeof(BehaviourProxy))]
    public class BehaviourProxyEditor : Editor
    {
        public override void OnInspectorGUI(){}
    }

    #endif

    public class BehaviourProxy : MonoBehaviour
    {
        public IBehaviour Parent;

        public void Awake() { if (Parent != null) Parent.MonoBehaviourAwake(); }
        public void Start() { if (Parent != null) Parent.Start(); }
        public void Update() { if (Parent != null) Parent.Update(); }
        public void FixedUpdate() { if (Parent != null) Parent.FixedUpdate(); }
    }

    public interface IBehaviour
    {
        void MonoBehaviourAwake();
        void Start();
        void Update();
        void FixedUpdate();
    }

    public class BehaviourScriptableObject : ScriptableObject, IBehaviour
    {
        public void Awake() { ScriptableObjectAwake(); }
        public virtual void ScriptableObjectAwake() { }
        public virtual void MonoBehaviourAwake() { }
        public virtual void Start() { }
        public virtual void Update() { }
        public virtual void FixedUpdate() { }
    }
}
```

Here there is an example GameManager singleton class using the SingletonScriptableObject (with a lot of comments):

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

//this attribute is optional but recommended. It will allow the creation of the singleton via
the asset menu.
//the singleton asset should be on the Resources folder.
[CreateAssetMenu(fileName = "GameManager", menuName = "Game Manager", order = 0)]
public class GameManager : SingletonScriptableObject<GameManager> {

    //any properties as usual
    public int Lives;
```

```
    public int Points;

    //optional (but recommended)
    //this method will run before the first scene is loaded. Initializing the singleton here
    //will allow it to be ready before any other GameObjects on every scene and will
    //will prevent the "initialization on first usage".
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    public static void BeforeSceneLoad() { BuildSingletonInstance(); }

    //optional,
    //will run when the Singleton Scriptable Object is first created on the assets.
    //Usually this happens on edit mode, not runtime. (the override keyword is mandatory for
this to work)
    public override void ScriptableObjectAwake(){
        Debug.Log(GetType().Name + " created." );
    }

    //optional,
    //will run when the associated MonoBehavioir awakes. (the override keyword is mandatory
for this to work)
    public override void MonoBehaviourAwake(){
        Debug.Log(GetType().Name + " behaviour awake." );

        //A coroutine example:
        //Singleton Objects do not have coroutines.
        //if you need to use coroutines use the atached MonoBehaviour
        Behaviour.StartCoroutine(SimpleCoroutine());
    }

    //any methods as usual
    private IEnumerator SimpleCoroutine(){
        while(true){
            Debug.Log(GetType().Name + " coroutine step." );
            yield return new WaitForSeconds(3);
        }
    }

    //optional,
    //Classic runtime Update method (the override keyword is mandatory for this to work).
    public override void Update(){

    }

    //optional,
    //Classic runtime FixedUpdate method (the override keyword is mandatory for this to work).
    public override void FixedUpdate(){

    }
}

/*
*  Notes:
*  - Remember that you have to create the singleton asset on edit mode before using it. You
have to put it on the Resources folder and of course it should be only one.
*  - Like other Unity Singleton this one is accessible anywhere in your code using the
"Instance" property i.e: GameManager.Instance
*/
```

Read Singletons in Unity online: https://riptutorial.com/unity3d/topic/2137/singletons-in-unity

Chapter 32: Tags

Introduction

A tag is a string that can be applied to mark `GameObject` types. In this way, it makes it easier to identify particular `GameObject` objects via code.

A tag can be applied to one or more game objects, but a game object will always only have one tag. By default, the tag *"Untagged"* is used to represent a `GameObject` that has not been intentionally tagged.

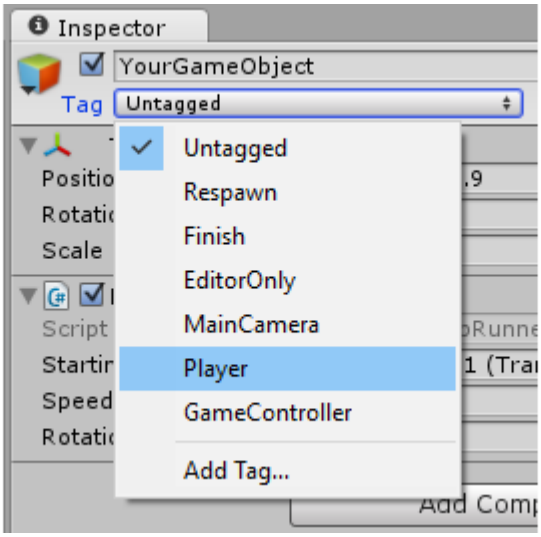
Examples

Creating and Applying Tags

Tags are typically applied via the editor; however, you can also apply tags via script. Any custom tag must be created via the *Tags & Layers* window before being applied to a game object.

Setting Tags in the Editor

With one or more game objects selected, you can select a tag from the inspector. Game objects will always carry a single tag; by default, game objects will be tagged as *"Untagged"*. You can also move to the *Tags & Layers* window, by selecting *"Add Tag..."*; however, it is important to note that this only takes you to the *Tags & Layers* window. Any tag you create will *not* automatically apply to the game object.



Setting Tags via Script

You can directly change a game objects tag via code. It is important to note that you *must* provide

a tag from the list of current tags; if you supply a tag that has not already been created, this will result in an error.

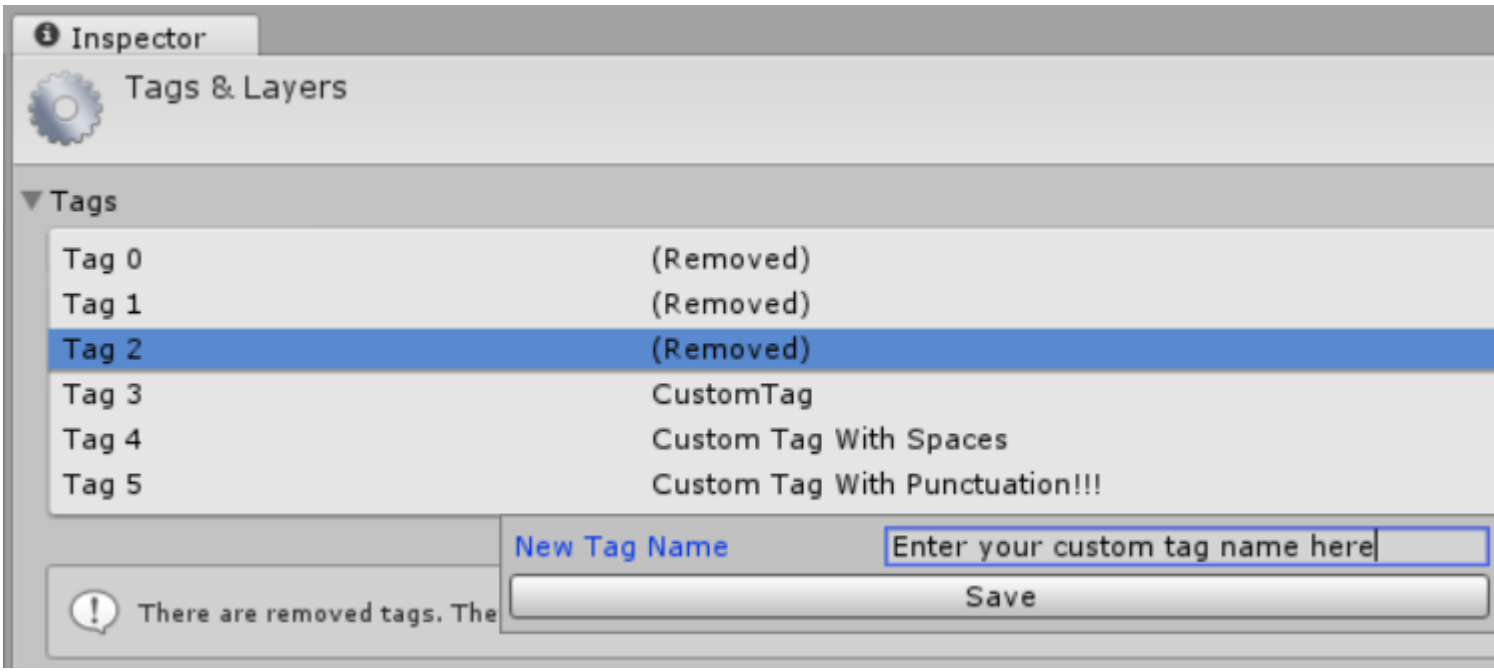
As detailed in other examples, using a series of `static string` variables as opposed to manually writing each tag can ensure consistency and reliability.

The following script demonstrates how we might change a series of game objects tags, using `static string` references to ensure consistency. Note the assumption that each `static string` represents a tag that has already been created in the *Tags & Layers* window.

```
using UnityEngine;

public class Tagging : MonoBehaviour
{
    static string tagUntagged = "Untagged";
    static string tagPlayer = "Player";
    static string tagEnemy = "Enemy";

    /// <summary>Represents the player character. This game object should
    /// be linked up via the inspector.</summary>
    public GameObject player;
    /// <summary>Represents all the enemy characters. All enemies should
    /// be added to the array via the inspector.</summary>
    public GameObject[] enemy;

    void Start ()
    {
        // We ensure that the game object this script is attached to
        // is left untagged by using the default "Untagged" tag.
        gameObject.tag = tagUntagged;

        // We ensure the player has the player tag.
        player.tag = tagUntagged;

        // We loop through the enemy array to ensure they are all tagged.
        for(int i = 0; i < enemy.Length; i++)
        {
            enemy[i].tag = tagEnemy;
        }
    }
}
```

Creating Custom Tags

Regardless of whether you set tags via the Inspector, or via script, tags *must* be declared via the *Tags & Layers* window before use. You can access this window by selecting *"Add Tags..."* from a game objects tag drop down menu. Alternatively, you can find the window under **Edit > Project Settings > Tags and Layers**.



Simply select the `+` button, enter the desired name and select `Save` to create a tag. Selecting the `-` button will remove the currently highlighted tag. Note that in this manner, the tag will be immediately displayed as *"(Removed)"*, and will be completely removed when the project is next reloaded.

Selecting the gear/cog from the top right of the window will allow you to reset all custom options. This will immediately remove all custom tags, along with any custom layer you may have under *"Sorting Layers"* and *"Layers"*.

Finding GameObjects by Tag:

Tags make it particularly easy to locate specific game objects. We can look for a single game object, or look for multiple.

Finding a Single `GameObject`

We can use the static function `GameObject.FindGameObjectWithTag(string tag)` to look for individual game objects. It is important to note that, in this way, game objects are not queried in any particular order. If you search for a tag that is used on multiple game objects in the scene, this function will not be able to guarantee *which* game object is returned. As such, it is more appropriate when we know that only *one* game object uses such tag, or when we are not worried about the exact instance of `GameObject` that is returned.

```
///<summary>We create a static string to allow us consistency.</summary>
string playerTag = "Player"

///<summary>We can now use the tag to reference our player GameObject.</summary>
GameObject player = GameObject.FindGameObjectWithTag(playerTag);
```

Finding an Array of `GameObject` instances

We can use the static function `GameObject.FindGameObjectsWithTag(string tag)` to look for *all* game objects that use a particular tag. This is useful when we want iterate through a group of particular game objects. This can also be useful if we want to find a *single* game object, but may have *multiple* game objects using the same tag. As we can not guarantee the exact instance returned by `GameObject.FindGameObjectWithTag(string tag)`, we must instead retrieve an array of all potential `GameObject` instances with `GameObject.FindGameObjectsWithTag(string tag)`, and further analyse the resulting array to find the instance we are looking for.

```
///<summary>We create a static string to allow us consistency.</summary>
string enemyTag = "Enemy";

///<summary>We can now use the tag to create an array of all enemy GameObjects.</summary>
GameObject[] enemies = GameObject.FindGameObjectsWithTag(enemyTag );

// We can now freely iterate through our array of enemies
foreach(GameObject enemy in enemies)
{
    // Do something to each enemy (link up a reference, check for damage, etc.)
}
```

Comparing Tags

When comparing two GameObjects by Tags, it should be noted that the following would cause Garbage Collector overhead as a string is created everytime:

```
if (go.Tag == "myTag")
{
    //Stuff
}
```

When performing those comparisons inside Update() and other regular Unity's callback (or a loop), you should use this heap allocation-free method:

```
if (go.CompareTag("myTag")
{
    //Stuff
}
```

Additionally it's easier to keep your tags in a static class.

```
public static class Tags
{
    public const string Player = "Player";
    public const string MyCustomTag = "MyCustomTag";
}
```

Then you can compare safely

```
if (go.CompareTag(Tags.MyCustomTag)
{
    //Stuff
}
```

this way, your tag strings are generated at compile time, and you limit the implications of spelling mistakes.

Just like keeping tags into a static class, it is also possible to store it into an enumeration:

```
public enum Tags
{
    Player, Ennemies, MyCustomTag;
}
```

and then you can compare it using the enum `toString()` method:

```
if (go.CompareTag(Tags.MyCustomTag.toString())
{
    //Stuff
}
```

Read Tags online: https://riptutorial.com/unity3d/topic/5534/tags

Chapter 33: Transforms

Syntax

- void Transform.Translate(Vector3 translation, Space relativeTo = Space.Self)
- void Transform.Translate(float x, float y, float z, Space relativeTo = Space.Self)
- void Transform.Rotate(Vector3 eulerAngles, Space relativeTo = Space.Self)
- void Transform.Rotate(float xAngle, float yAngle, float zAngle, Space relativeTo = Space.Self)
- void Transform.Rotate(Vector3 axis, float angle, Space relativeTo = Space.Self)
- void Transform.RotateAround(Vector3 point, Vector3 axis, float angle)
- void Transform.LookAt(Transform target, Vector3 worldUp = Vector3.up)
- void Transform.LookAt(Vector3 worldPosition, Vector3 worldUp = Vector3.up)

Examples

Overview

Transforms hold the majority of data about an object in unity, including it's parent(s), child(s), position, rotation, and scale. It also has functions to modify each of these properties. Every GameObject has a Transform.

Translating (moving) an object

```
// Move an object 10 units in the positive x direction
transform.Translate(10, 0, 0);

// translating with a vector3
vector3 distanceToMove = new Vector3(5, 2, 0);
transform.Translate(distanceToMove);
```

Rotating an object

```
// Rotate an object 45 degrees about the Y axis
transform.Rotate(0, 45, 0);

// Rotates an object about the axis passing through point (in world coordinates) by angle in
degrees
transform.RotateAround(point, axis, angle);
// Rotates on it's place, on the Y axis, with 90 degrees per second
transform.RotateAround(Vector3.zero, Vector3.up, 90 * Time.deltaTime);

// Rotates an object to make it's forward vector point towards the other object
transform.LookAt(otherTransform);
// Rotates an object to make it's forward vector point towards the given position (in world
coordinates)
transform.LookAt(new Vector3(10, 5, 0));
```

More information and examples can be seen at Unity documentation.

Also note that if the game is using rigid bodies, then the transform should not be interacted with directly (unless the rigid body has `isKinematic == true`). In those case use AddForce or other similar methods to act on the rigid body directly.

Parenting and Children

Unity works with hierarchies in order to keep your project organized. You can assign objects a place in the hierarchy using the editor but you can also do this through code.

Parenting

You can set an object's parent with the following methods

```
var other = GetOtherGameObject();
other.transform.SetParent( transform );
other.transform.SetParent( transform, worldPositionStays );
```

Whenever you set a transforms parent, it will keep the objects position as a world position. You can choose to make this position relative by passing *false* for the *worldPositionStays* parameter.

You can also check if the object is a child of another transform with the following method

```
other.transform.IsChildOf( transform );
```

Getting a Child

Since objects can be parented to one another, you can also find children in the hierarchy. The simplest way of doing this is by using the following method

```
transform.Find( "other" );
transform.FindChild( "other" );
```

Note: FindChild calls Find under the hood

You can also search for children further down the hierarchy. You do this by adding in a "/" to specify going a level deeper.

```
transform.Find( "other/another" );
transform.FindChild( "other/another" );
```

Another way of fetching a child is using the GetChild

```
transform.GetChild( index );
```

GetChild requires an integer as index which must be smaller than the total child count

```
int count = transform.childCount;
```

Changing Sibling Index

You can change the order of the children of a GameObject. You can do this to define the draw order of the children (assuming that they are on the same Z level and the same sorting order).

```
other.transform.SetSiblingIndex( index );
```

You can also quickly set the sibling index to either first or last using the following methods

```
other.transform.SetAsFirstSibling();
other.transform.SetAsLastSibling();
```

Detaching all Children

If you want to release all children of a transform, you can do this:

```
foreach(Transform child in transform)
{
    child.parent = null;
}
```

Also, Unity provides a method for this purpose:

```
transform.DetachChildren();
```

Basically, both looping and `DetachChildren()` set the parents of first-depth children to null - which means they will have no parents.

(first-depth children: the transforms that are directly child of transform)

Read Transforms online: https://riptutorial.com/unity3d/topic/2190/transforms

Chapter 34: Unity Animation

Examples

Basic Animation for Running

This code shows a simple example of animation in Unity.

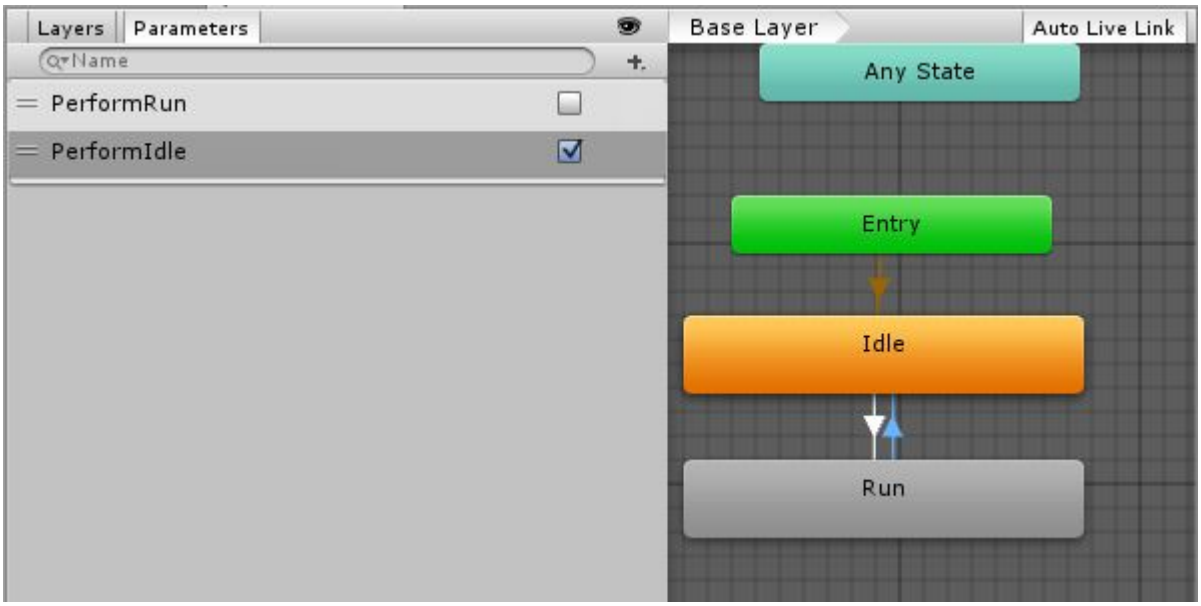
For this example, you should have 2 animation clips; Run and Idle. Those animations should be Stand-In-Place motions. Once the animation clips are selected, create an Animator Controller. Add this Controller to the player or game object you want to animate.

Open the Animator window from Windows option. Drag the 2 animation clips to the Animator window and 2 states would be created. Once created, use the left parameters tab to add 2 parameters, both of them as bool. Name one as "PerformRun" and other as "PerformIdle". Set "PerformIdle" to true.

Make transitionsfrom Idle state to Run and Run to idle (Refer the image). Click on Idle->Run transition and in the Inspector window, de-select HasExit. Do the same for the other transition. For Idle->Run transition, add a condition: PerformIdle. For Run->Idle, add a condition: PerformRun. Add the C# script given below to the game object. It should run with animation using the Up button and rotate with Left and Right buttons.

```
using UnityEngine;
using System.Collections;

public class RootMotion : MonoBehaviour {

//Public Variables
[Header("Transform Variables")]
public float RunSpeed = 0.1f;
public float TurnSpeed = 6.0f;


Animator animator;

void Start()
{
    /**
    * Initialize the animator that is attached on the current game object i.e. on which you
will attach this script.
    */
    animator = GetComponent<Animator>();
}

void Update()
{
    /**
    * The Update() function will get the bool parameters from the animator state machine and
set the values provided by the user.
    * Here, I have only added animation for Run and Idle. When the Up key is pressed, Run
animation is played. When we let go, Idle is played.
```

```
    */

    if (Input.GetKey (KeyCode.UpArrow)) {
            animator.SetBool ("PerformRun", true);
            animator.SetBool ("PerformIdle", false);
        } else {
            animator.SetBool ("PerformRun", false);
            animator.SetBool ("PerformIdle", true);
        }
}

void OnAnimatorMove()
    {
        /**
         * OnAnimatorMove() function will shadow the "Apply Root Motion" on the animator. Your
game objects psoition will now be determined
         * using this fucntion.
         */
        if (Input.GetKey (KeyCode.UpArrow)){
            transform.Translate (Vector3.forward * RunSpeed);
            if (Input.GetKey (KeyCode.RightArrow)) {
                transform.Rotate (Vector3.up * Time.deltaTime * TurnSpeed);
            }
            else if (Input.GetKey (KeyCode.LeftArrow)) {
                transform.Rotate (-Vector3.up * Time.deltaTime * TurnSpeed);
            }
        }

    }



}
```



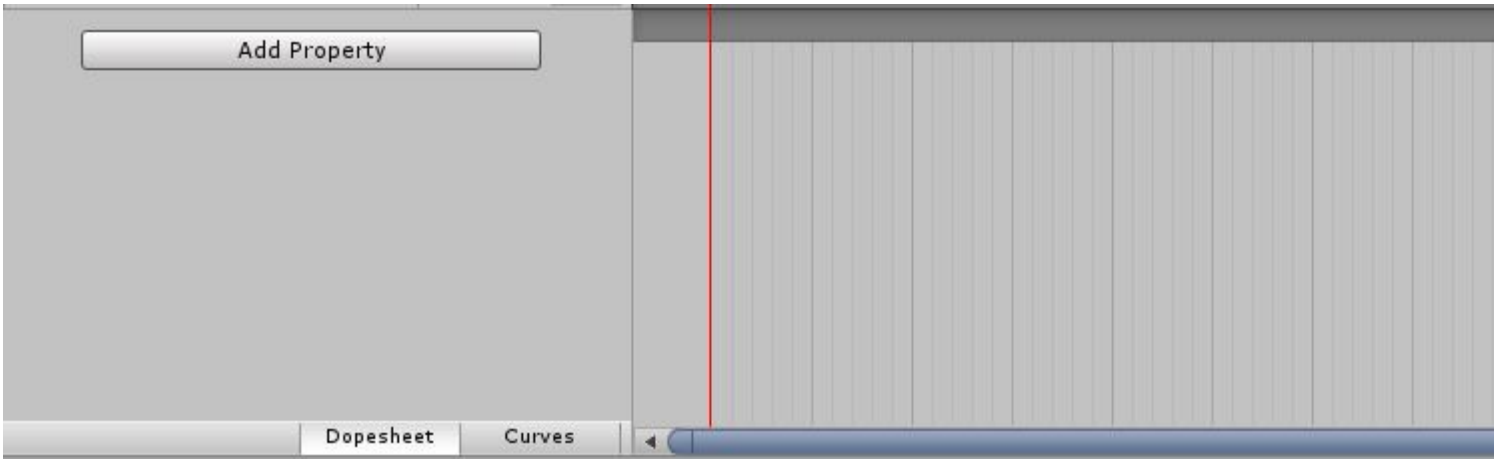
Creating and Using Animation Clips

This example will show how to make and use animation clips for game objects or players.

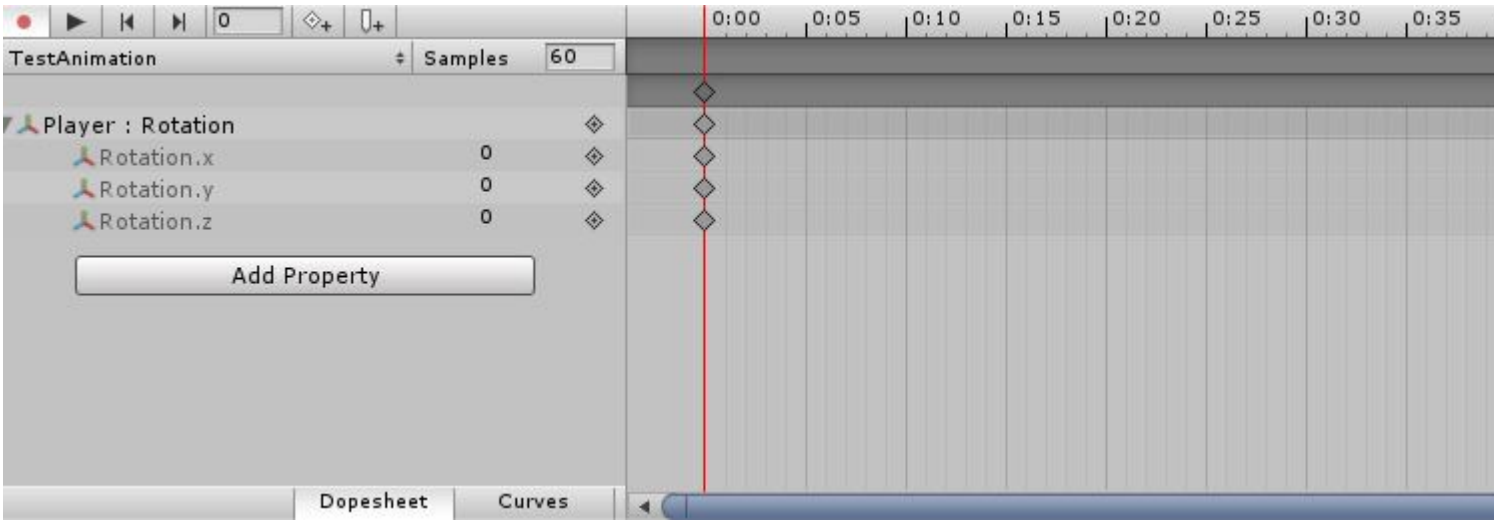
Note, the models used in this example are downloaded from Unity Asset Store. The player was downloaded from the following link: https://www.assetstore.unity3d.com/en/#!/content/21874.

To create animations, first open the Animation Window. You can open it by clicking on Window and Select Animation or press Ctrl+6. The select the game object to which you want to apply the animation clip, from the Hierarchy Window, and then click on Create button on the Animation Window.

Name your animation (like IdlePlayer, SprintPlayer, DyingPlayer etc.) and Save it. Now, from the Animation Window, click on Add Property button. This will allow you to change the property of the game object or player with respect to time. This can include Tranforms like rotation, position and scale and anyother property that is attached to the game object e.g. Collider, Mesh Renderer etc.



To create a running animation for game object, you will need a humanoid 3D model. You can download the model from the above link. Follow the above steps to create a new animation. Add a Transform property and select Rotation for one of the character leg.



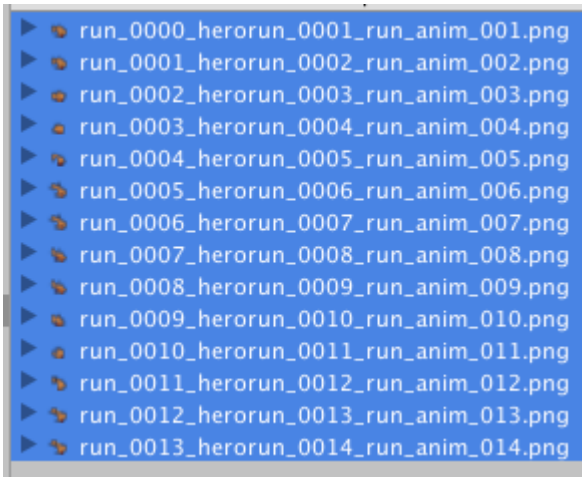
At this moment, your Play button and Rotation values in the game object property would have turned red. Click the drop down arrow to see the rotation X, Y and Z values. The default animation time is set to 1 second. Animations use key frames to interpolate between values. To animate, add keys at different points in time and change the rotation values from the Inspector Window. For e.g. the rotation value at time 0.0s can be 0.0. At time 0.5s the value can be 20.0 for X. At time 1.0s the value can be 0.0. We can end our animation at 1.0s.

Your animation length depends on the last Keys that you add to the Animation. You can add more keys to make the interpolation smoother.

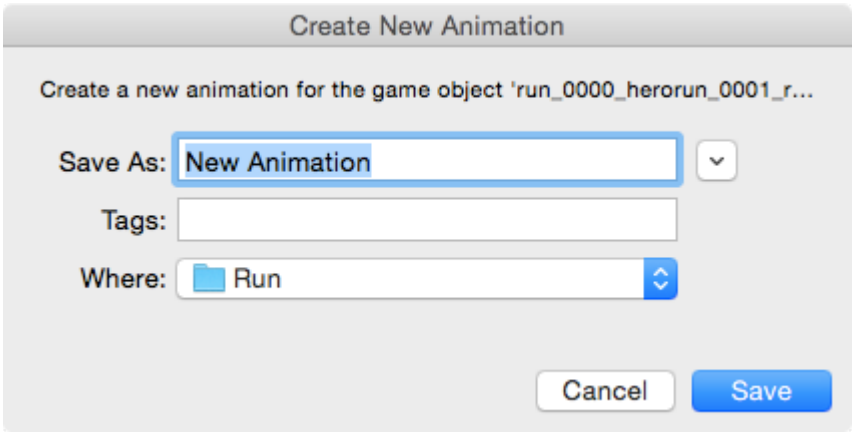
2D Sprite Animation

Sprite animation consists in showing an existing sequence of images or frames.

First import a sequence of images to the asset folder. Either create some images from scratch or download some from the Asset Store. (This example uses this free asset.)

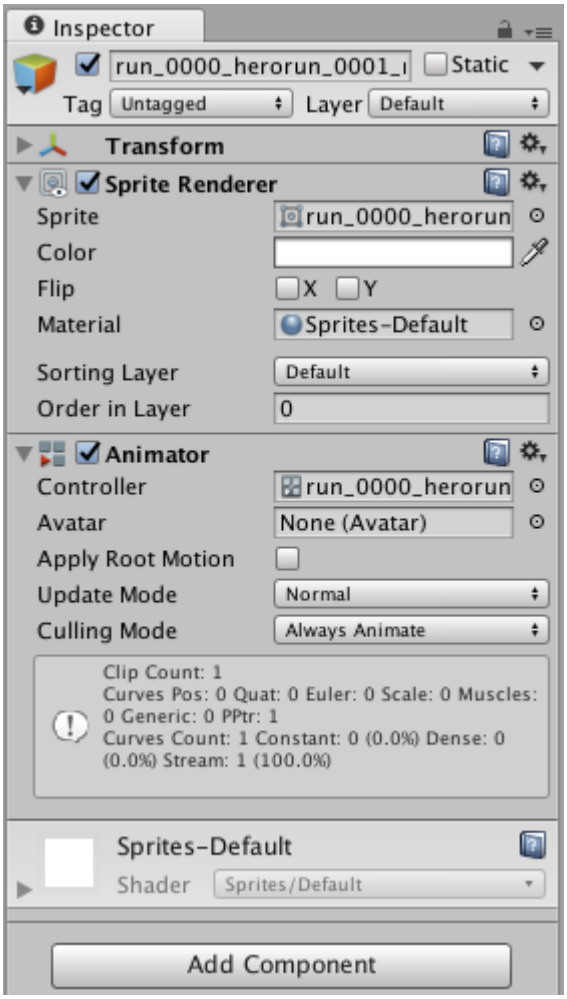


Drag every individual image of a single animation from the assets folder to the scene view. Unity will show a dialog for naming the new animation clip.

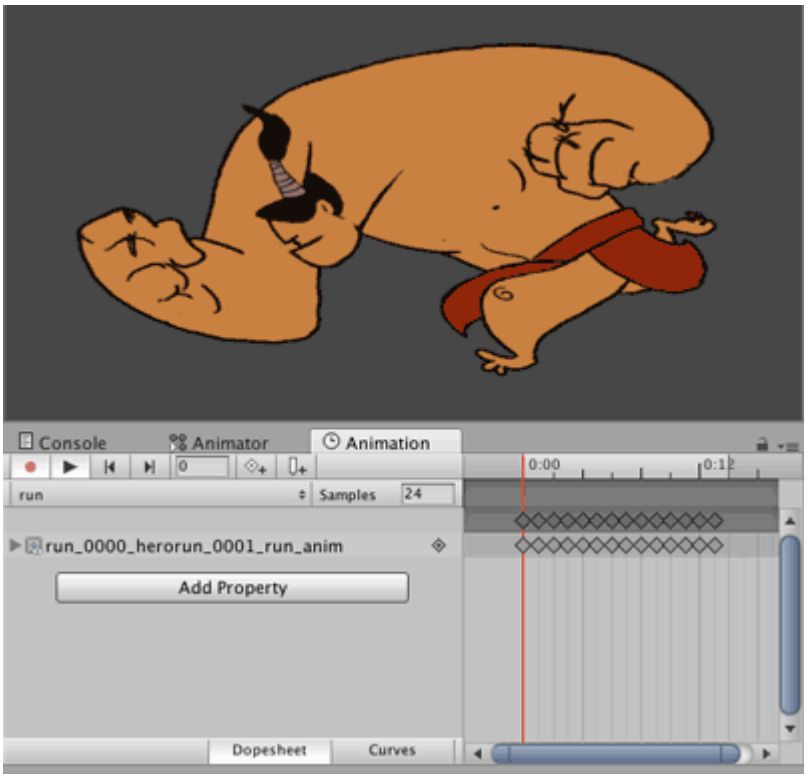


This is a useful shortcut for:

- creating new game objects
- assigning two components (a Sprite Renderer and an Animator)
- creating animation controllers (and linking the new Animator component to them)
- creating animation clips with the selected frames



Preview the playback in the animation tab by clicking Play:



The same method can be used to create new animations for the same game object, then deleting the new game object and animation controller. Add the new animation clip to the animation controller of that object in the same manner as with 3D animation.

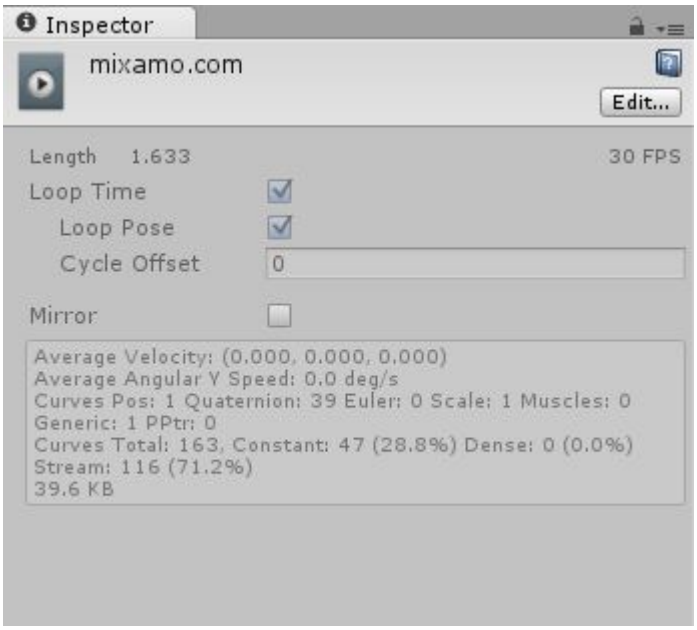
Animation Curves

Animation curves allows you to change a float parameter as the animation plays. For example, if there is an animation of length 60 seconds and you want a float value/parameter, call it X, to vary through the animation (like at animation time = 0.0s; X = 0.0 , at animation time = 30.0s; X = 1.0, at animation time = 60.0s; X = 0.0).

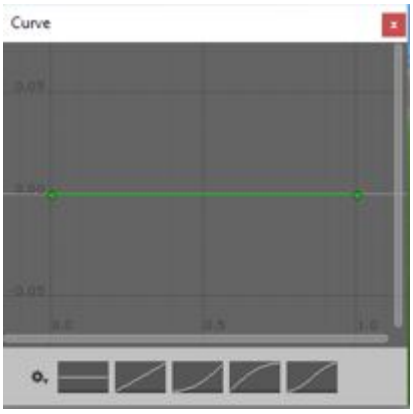
Once you have the float value, you can use it to translate, rotate, scale or use it in any other way.

For my example, I will show a player game object running. When the animation for run plays, the player's translation speed should increase as the animation proceeds. When the animation reaches its end, the translation speed should decrease.

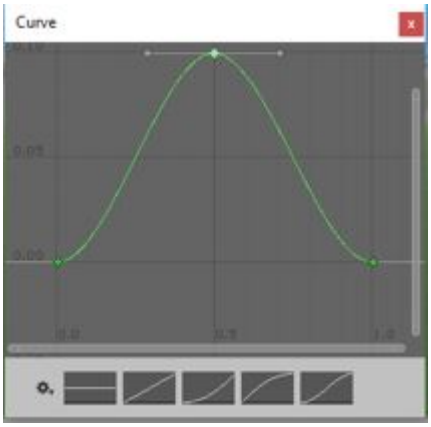
I have a running animation clip created. Select the clip and then in the inspector window, click on Edit.



Once there, scroll down to Curves. Click on the + sign to add a curve. Name the Curve e.g. ForwardRunCurve. Click on the miniature curve on the right. It will open a small window with a default curve in it.



We want a parabolic shaped curve where it rises and then falls. By default, there are 2 points on the line. You can add more points by double clicking on the curve. Drag the points to create a shape similar to the following.



In the Animator Window, add the running clip. Also, add a float parameter with the same name as the curve i.e. ForwardRunCurve.

When the Animation plays, the float value will change according to the curve. The following code will show how to use the float value:

```
using UnityEngine;
using System.Collections;

public class RunAnimation : MonoBehaviour {

Animator animator;
float curveValue;

void Start()
{
    animator = GetComponent<Animator>();
}

void Update()
{
    curveValue = animator.GetFloat("ForwardRunCurve");

    transform.Translate (Vector3.forward * curveValue);
}

}
```

The curveValue variable holds the value of the curve(ForwardRunCruve) at any given time. We are using that value to change the speed of the translation. You can attach this script to the player game object.

Read Unity Animation online: https://riptutorial.com/unity3d/topic/5448/unity-animation

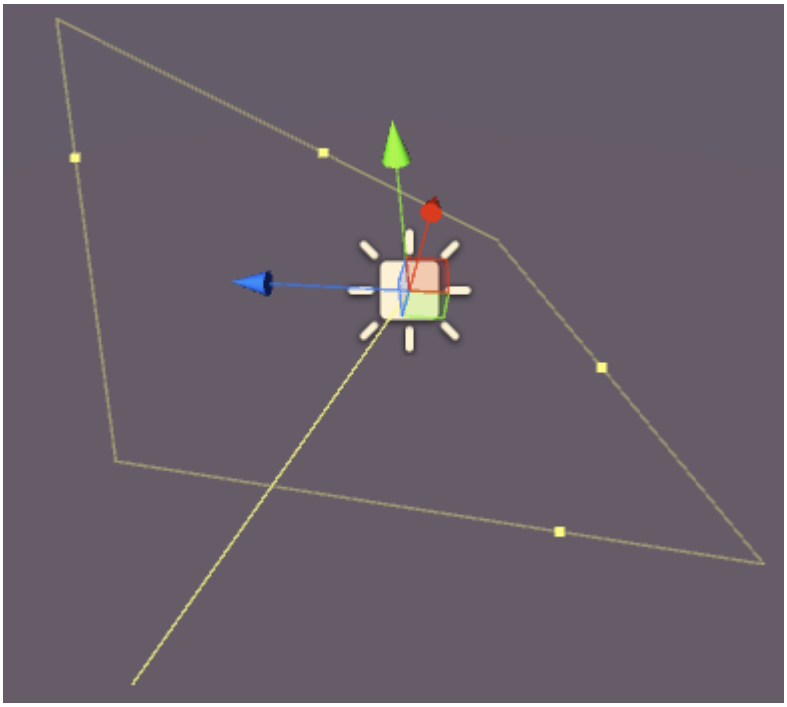
Chapter 35: Unity Lighting

Examples

Types of Light

Area Light

Light is emitted across the surface of a rectangular area. They are baked only which means you won't be able to see the effect until you bake the scene.

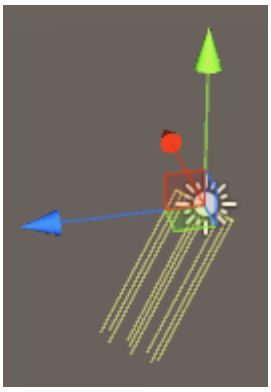


Area Lights have the following properties:

- **Width** - Width of light area.
- **Height** - Height of light area.
- **Color** - Assign the color of the light.
- **Intensity** - How powerful the light is from 0 - 8.
- **Bounce Intensity** - How powerful the *indirect* light is from 0 - 8.
- **Draw Halo** - Will draw a halo around the light.
- **Flare** - Allows you to assign a flare effect to the light.
- **Render Mode** - Auto, Important, Not Important.
- **Culling Mask** - Allows you to selectively light parts of a scene.

Directional Light

Directional Lights emit light in a single direction (much like the sun). It does not matter where in the scene the actual GameObject is placed as the light is "everywhere". The light intensity does not diminish like the other light types.

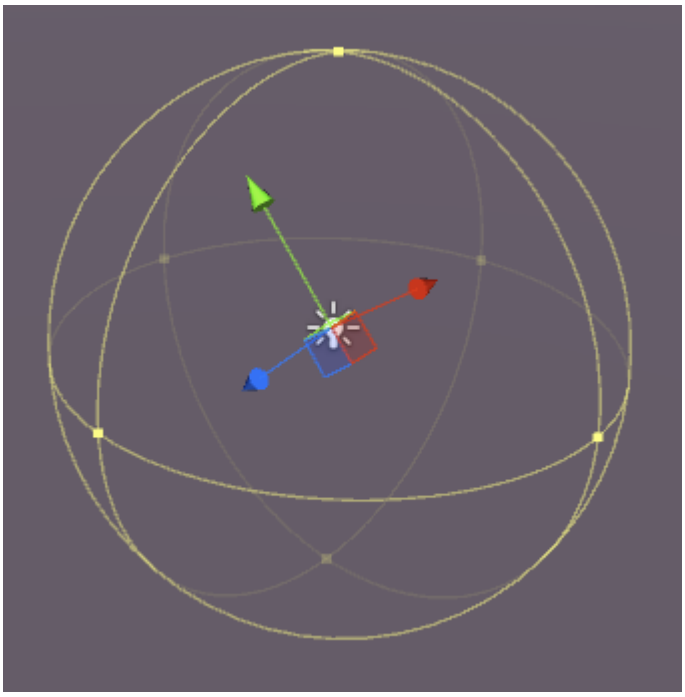


A Directional Light has the following properties:

- **Baking** - Realtime, Baked or Mixed.
- **Color** - Assign the color of the light.
- **Intensity** - How powerful the light is from 0 - 8.
- **Bounce Intensity** - How powerful the *indirect* light is from 0 - 8.
- **Shadow Type** - No Shadows, Hard Shadows or Soft Shadows.
- **Cookie** - Allow you to assign a cookie for the light.
- **Cookie Size** - The size of the assigned cookie.
- **Draw Halo** - Will draw a halo around the light.
- **Flare** - Allows you to assign a flare effect to the light.
- **Render Mode** - Auto, Important, Not Important.
- **Culling Mask** - Allows you to selectively light parts of a scene.

Point Light

A Point Light emits light from a point in space in all directions. The further from the origin point, the less intense the light.

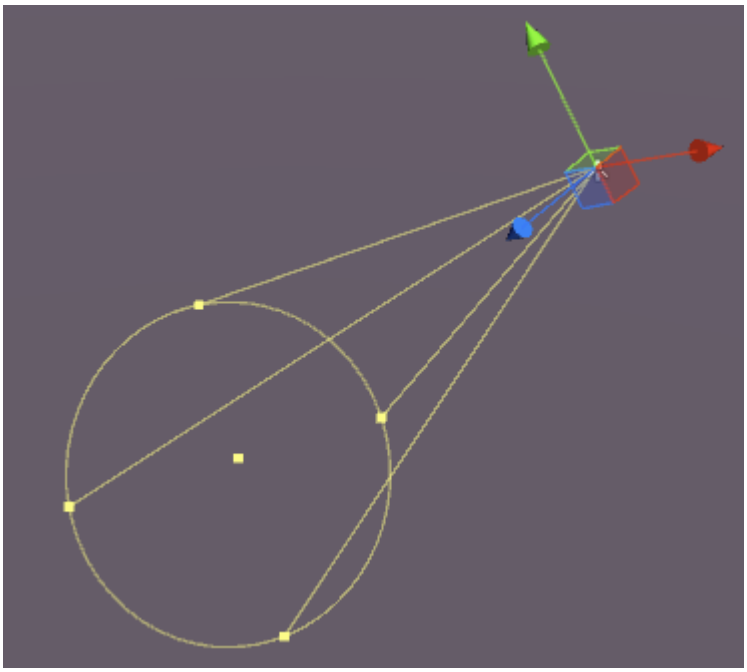


Point Lights have the following properties:

- **Baking** - Realtime, Baked or Mixed.
- **Range** - The distance from the point where light no longer reaches.
- **Color** - Assign the color of the light.
- **Intensity** - How powerful the light is from 0 - 8.
- **Bounce Intensity** - How powerful the *indirect* light is from 0 - 8.
- **Shadow Type** - No Shadows, Hard Shadows or Soft Shadows.
- **Cookie** - Allow you to assign a cookie for the light.
- **Draw Halo** - Will draw a halo around the light.
- **Flare** - Allows you to assign a flare effect to the light.
- **Render Mode** - Auto, Important, Not Important.
- **Culling Mask** - Allows you to selectively light parts of a scene.

Spot Light

A Spot Light is much like a Point Light but the emission is restricted to an angle. The result is a "cone" of light, useful for car headlights or searchlights.



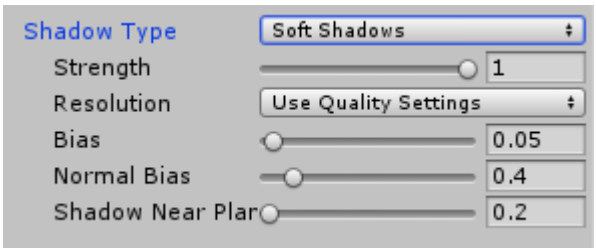
Spot Lights have the following properties:

- **Baking** - Realtime, Baked or Mixed.
- **Range** - The distance from the point where light no longer reaches.
- **Spot Angle** - The angle of light emission.
- **Color** - Assign the color of the light.
- **Intensity** - How powerful the light is from 0 - 8.
- **Bounce Intensity** - How powerful the *indirect* light is from 0 - 8.
- **Shadow Type** - No Shadows, Hard Shadows or Soft Shadows.
- **Cookie** - Allow you to assign a cookie for the light.
- **Draw Halo** - Will draw a halo around the light.
- **Flare** - Allows you to assign a flare effect to the light.
- **Render Mode** - Auto, Important, Not Important.
- **Culling Mask** - Allows you to selectively light parts of a scene.

Note about Shadows

If you select Hard or Soft Shadows, the following options become available in the inspector:

- **Strength** - How dark the shadows are from 0 - 1.
- **Resolution** - How detailed shadows are.
- **Bias** - he degree to which shadow casting surfaces are pushed away from the light.
- **Normal Bias** - The degree to which shadow casting surfaces are pushed inwards along their normals.
- **Shadow Near Plane** - 0.1 - 10.



Emission

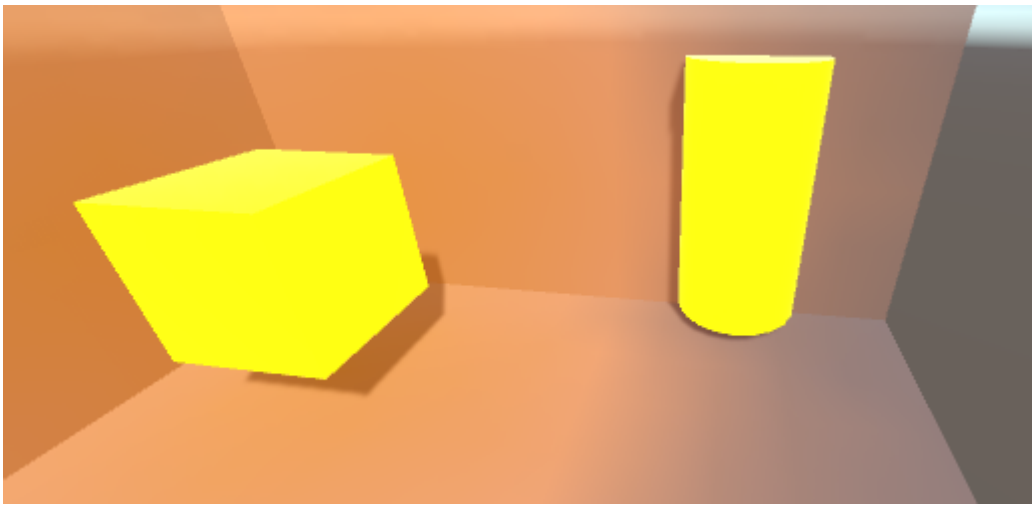
Emission is when a surface (or rather a material) emits light. In the inspector panel for a material on a static object using the Standard Shader there is an emission property:

If you change this property to a value higher than the default of 0, you can set the emission color, or assign an **emission map** to the material. Any texture assigned to this slot will enable the emission to use its own colors.

There is also a Global Illumination option which allows you to set whether the emission is baked onto nearby static objects or not:

- **Baked** - The emission will be baked into the scene
- **Realtime** - The emission will affect dynamic objects
- **None** - The emission will not affect nearby objects

If the object is *not* set to static, the effect will still make the object appear to "glow" but no light is emitted. The cube here is static, the cylinder is not:



You can set the emission color in code like this:

```
Renderer renderer = GetComponent<Renderer>();
Material mat = renderer.material;
mat.SetColor("_EmissionColor", Color.yellow);
```

Light emitted will fall off at a quadratic rate and will only show against static materials in the scene.

Read Unity Lighting online: https://riptutorial.com/unity3d/topic/7884/unity-lighting

Chapter 36: Unity Profiler

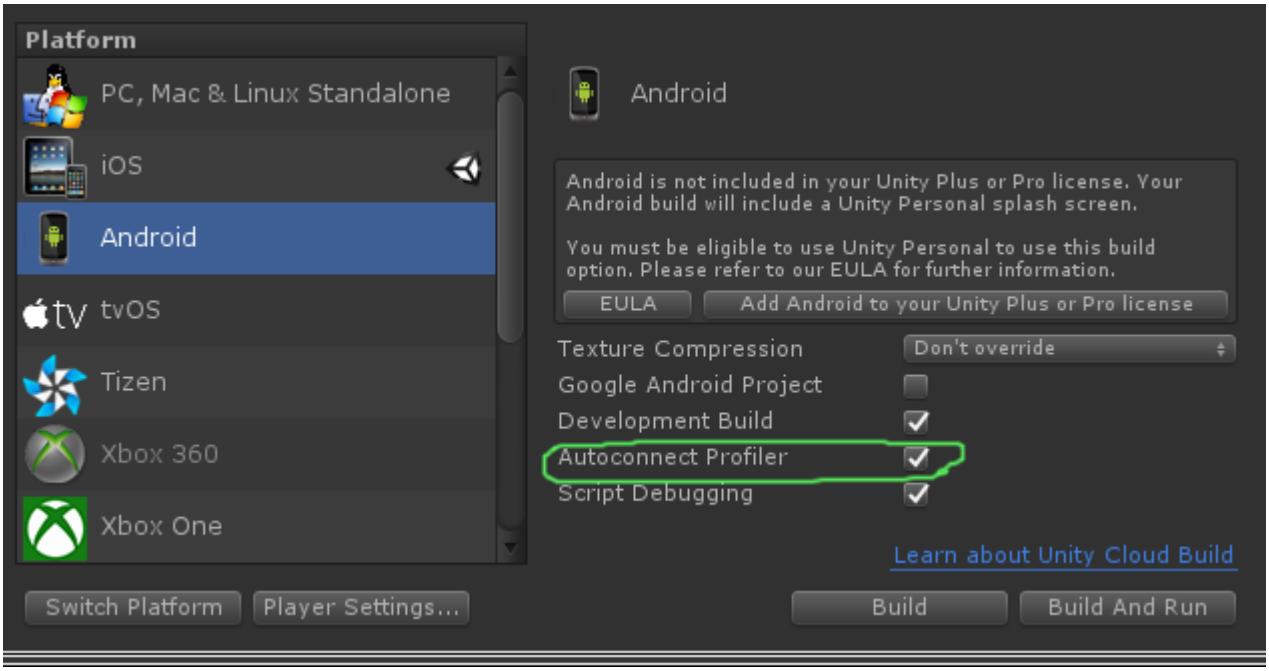
Remarks

Using Profiler on different Device

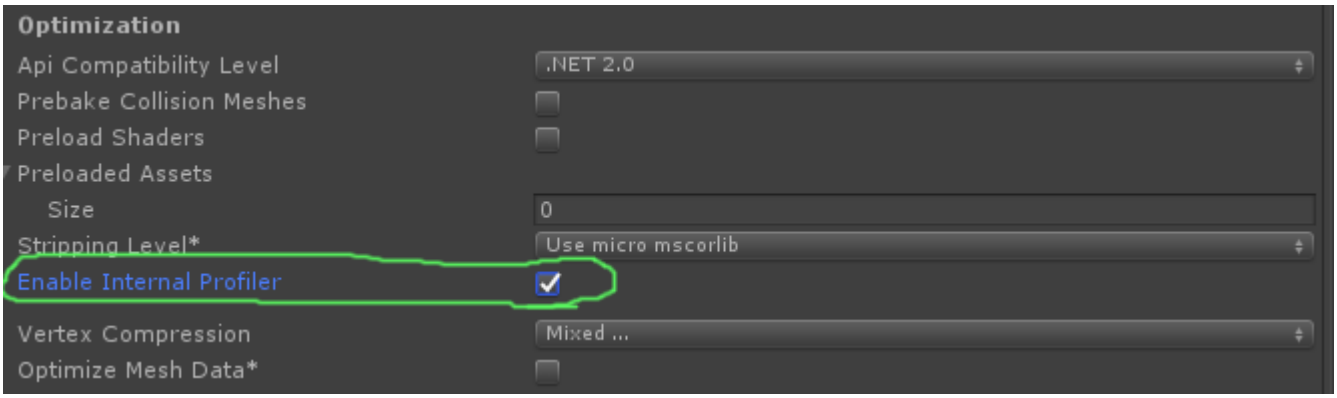
There are few importants things to know to properly hook the Profiler on different platforms.

Android

In order to properly attach the profile, "Build and Run" button from the Build Settings window with the option **Autoconnect Profiler** checked must be used.



Another mandatory option, in Android Player Settings inspector in the Other Settings tab, there is a checkbox **Enable Internal profiler** which needs to be checked so LogCat will output profiler info.

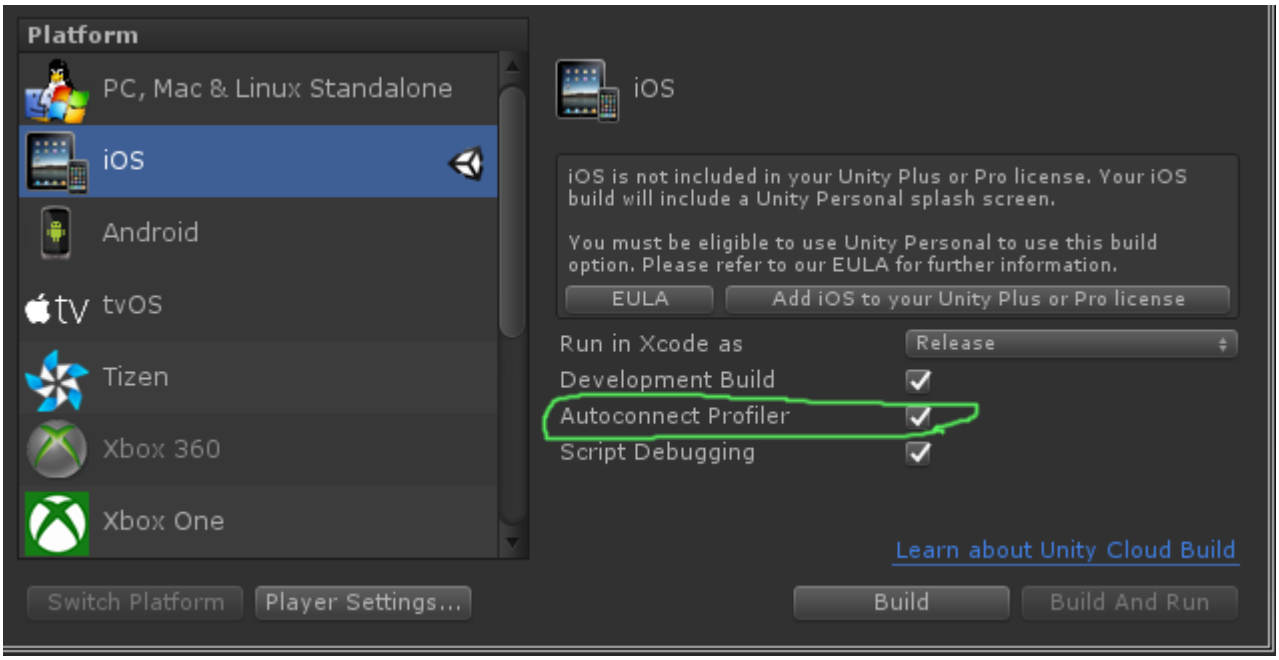


Using only "Build" will not allow the profiler to connect to an Android device because the "Build

and Run" use specific command line arguments to start it with LogCat.

iOS

In order to properly attach the profile, "Build and Run" button from the Build Settings window with the option **Autoconnect Profiler** checked must be used on the first run.



On iOS, there is no option in player settings that must be set for the Profiler to be enable. It should work out of the box.

Examples

Profiler Markup

Using the Profiler Class

One very good practice is to use Profiler.BeginSample and Profiler.EndSample because it will have its own entry in the Profiler Window.

Also, those tag will be stripped out on non-Development build using using ConditionalAttribute, so you don't need to remove them from your code.

```
public class SomeClass : MonoBehaviour
{
    void SomeFunction()
    {
        Profiler.BeginSample("SomeClass.SomeFunction");
        // Various call made here
        Profiler.EndSample();
    }
}
```

This will create an Entry "SomeClass.SomeFunction" in the Profiler Window that will allow easier debugging and identification of Bottle neck.

Read Unity Profiler online: https://riptutorial.com/unity3d/topic/6974/unity-profiler

Chapter 37: User Interface System (UI)

Examples

Subscribing to event in code

By default, one should subscribe to event using inspector, but sometimes it's better to do it in code. In this example we subscribe to click event of a button in order to handle it.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Button))]
public class AutomaticClickHandler : MonoBehaviour
{
    private void Awake()
    {
        var button = this.GetComponent<Button>();
        button.onClick.AddListener(HandleClick);
    }

    private void HandleClick()
    {
        Debug.Log("AutomaticClickHandler.HandleClick()", this);
    }
}
```

The UI components usually provide their main listener easily :

- Button : onClick
- Dropdown : onValueChanged
- InputField : onEndEdit, onValidateInput, onValueChanged
- Scrollbar : onValueChanged
- ScrollRect : onValueChanged
- Slider : onValueChanged
- Toggle : onValueChanged

Adding mouse listeners

Sometimes, you want to add listeners on particular events not natively provided by the components, in particular mouse events. To do so, you will have to add them by yourself using an `EventTrigger` component :

```
using UnityEngine;
using UnityEngine.EventSystems;

[RequireComponent(typeof( EventTrigger ))]
public class CustomListenersExample : MonoBehaviour
{
    void Start( )
```

```
    {
        EventTrigger eventTrigger = GetComponent<EventTrigger>( );
        EventTrigger.Entry entry = new EventTrigger.Entry( );
        entry.eventID = EventTriggerType.PointerDown;
        entry.callback.AddListener( ( data ) => { OnPointerDownDelegate(
(PointerEventData)data ); } );
        eventTrigger.triggers.Add( entry );
    }

    public void OnPointerDownDelegate( PointerEventData data )
    {
        Debug.Log( "OnPointerDownDelegate called." );
    }
}
```

Various eventID are possible :

- PointerEnter
- PointerExit
- PointerDown
- PointerUp
- PointerClick
- Drag
- Drop
- Scroll
- UpdateSelected
- Select
- Deselect
- Move
- InitializePotentialDrag
- BeginDrag
- EndDrag
- Submit
- Cancel

Read User Interface System (UI) online: https://riptutorial.com/unity3d/topic/2296/user-interface-system--ui-

Chapter 38: Using Git source control with Unity

Examples

Using Git Large File Storage (LFS) with Unity

Foreword

Git can work with video game development out of the box. However the main caveat is that versioning large (>5 MB) media files can be a problem over the long term as your commit history bloats - Git simply wasn't originally built for versioning binary files.

The great news is that since mid-2015 GitHub has released a plugin for Git called Git LFS that directly deals with this problem. You can now easily and efficiently version large binary files!

Finally, this documentation is focused on the specific requirements and information necessary to ensure your Git life works well with video game development. This guide will not cover how to use Git itself.

Installing Git & Git-LFS

You have a number of options available to you as a developer and the first choice is whether to install the core Git command-line or let one of the popular Git GUI applications deal with it for you.

Option 1: Use a Git GUI Application

This is really a personal preference here as there are quite a few options in terms of Git GUI or whether to use a GUI at all. You have a number of applications to choose from, here are 3 of the more popular ones:

- Sourcetree (Free)
- Github Desktop (Free)
- SmartGit (Commerical)

Once you've installed your application of choice, please google and follow instructions on how to ensure it is setup for Git-LFS. We'll be skipping this step in this guide as it is application specific.

Option 2: Install Git & Git-LFS

This is pretty simple - Install Git. Then. Install Git LFS.

Configuring Git Large File Storage on your project

If you're using the Git LFS plugin to give better support for binary files, then you'll need to set some file types to be managed by Git LFS. Add the below to your `.gitattributes` file in the root of your repository to support common binary files used in Unity projects:

```
# Image formats:
*.tga filter=lfs diff=lfs merge=lfs -text
*.png filter=lfs diff=lfs merge=lfs -text
*.tif filter=lfs diff=lfs merge=lfs -text
*.jpg filter=lfs diff=lfs merge=lfs -text
*.gif filter=lfs diff=lfs merge=lfs -text
*.psd filter=lfs diff=lfs merge=lfs -text

# Audio formats:
*.mp3 filter=lfs diff=lfs merge=lfs -text
*.wav filter=lfs diff=lfs merge=lfs -text
*.aiff filter=lfs diff=lfs merge=lfs -text

# 3D model formats:
*.fbx filter=lfs diff=lfs merge=lfs -text
*.obj filter=lfs diff=lfs merge=lfs -text

# Unity formats:
*.sbsar filter=lfs diff=lfs merge=lfs -text
*.unity filter=lfs diff=lfs merge=lfs -text

# Other binary formats
*.dll filter=lfs diff=lfs merge=lfs -text
```

Setting up a Git repository for Unity

When initializing a Git repository for Unity development, there are a couple of things that need to be done.

Unity Ignore Folders

Not everything should be versioned in the repository. You can add the template below to your `.gitignore` file in the root of your repository. Or alternatively, you can check the open source Unity .gitignore on GitHub and alternatively generate one using gitignore.io for unity.

```
# Unity Generated
[Tt]emp/
[Ll]ibrary/
[Oo]bj/

# Unity3D Generated File On Crash Reports
sysinfo.txt
```

```
# Visual Studio / MonoDevelop Generated
ExportedObj/
obj/
*.csproj
*.unityproj
*.sln
*.suo
*.tmp
*.user
*.userprefs
*.pidb
*.booproj
*.svd

# OS Generated
desktop.ini
.DS_Store
.DS_Store?
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db
```

To learn more about how to setup a .gitignore file, check out here.

Unity Project Settings

By default Unity projects aren't setup to support versioning correctly.

1. (Skip this step in v4.5 and up) Enable `External` option in `Unity → Preferences → Packages → Repository`.
2. Switch to `Visible Meta Files` in `Edit → Project Settings → Editor → Version Control Mode`.
3. Switch to `Force Text` in `Edit → Project Settings → Editor → Asset Serialization Mode`.
4. Save the scene and project from `File` menu.

Additional Configuration

One of the few major annoyances one has with using Git with Unity projects is that Git doesn't care about directories and will happily leave empty directories around after removing files from them. Unity will make `*.meta` files for these directories and can cause a bit of a battle between team members when Git commits keep adding and removing these meta files.

Add this Git post-merge hook to the `/.git/hooks/` folder for repositories with Unity projects in them. After any Git pull/merge, it will look at what files have been removed, check if the directory it existed in is empty, and if so delete it.

Scenes and Prefabs merging

A common problem when working with Unity is when 2 or more developers are modifying a Unity scene or prefab (*.unity files). Git does not know how to merge them correctly out of the box.

Thankfully the Unity team deployed a tool called SmartMerge which makes simple merge automatic. The first thing to do is to add the following lines to your `.git` or `.gitconfig` file: (Windows: `%USERPROFILE%\.gitconfig`, Linux/Mac OS X: `~/.gitconfig`)

```
[merge]
tool = unityyamlmerge

[mergetool "unityyamlmerge"]
trustExitCode = false
cmd = '<path to UnityYAMLMerge>' merge -p "$BASE" "$REMOTE" "$LOCAL" "$MERGED"
```

On **Windows** the path to UnityYAMLMerge is :

```
C:\Program Files\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

or

```
C:\Program Files (x86)\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

and on **MacOSX** :

```
/Applications/Unity/Unity.app/Contents/Tools/UnityYAMLMerge
```

Once this is done, the mergetool will be available when conflicts arise during merge/rebase. Don't forget to run `git mergetool` manually to trigger UnityYAMLMerge.

Read Using Git source control with Unity online: https://riptutorial.com/unity3d/topic/2195/using-git-source-control-with-unity

Chapter 39: Vector3

Introduction

The `Vector3` structure represents a 3D coordinate, and is one of the backbone structures of the `UnityEngine` library. The `Vector3` structure is most commonly found in the `Transform` component of most game objects, where it is used to hold *position* and *scale*. `Vector3` provides good functionality for performing common vector operations. You can read more on the `Vector3` structure in the Unity API.

Syntax

- public Vector3();
- public Vector3(float x, float y);
- public Vector3(float x, float y, float z);
- Vector3.Lerp(Vector3 startPosition, Vector3 targetPosition, float movementFraction);
- Vector3.LerpUnclamped(Vector3 startPosition, Vector3 targetPosition, float movementFraction);
- Vector3.MoveTowards(Vector3 startPosition, Vector3 targetPosition, float distance);

Examples

Static Values

The `Vector3` structure contains some static variables that provide commonly used `Vector3` values. Most represent a *direction*, but they can still be used creatively to provide additional functionality.

`Vector3.zero` and `Vector3.one`

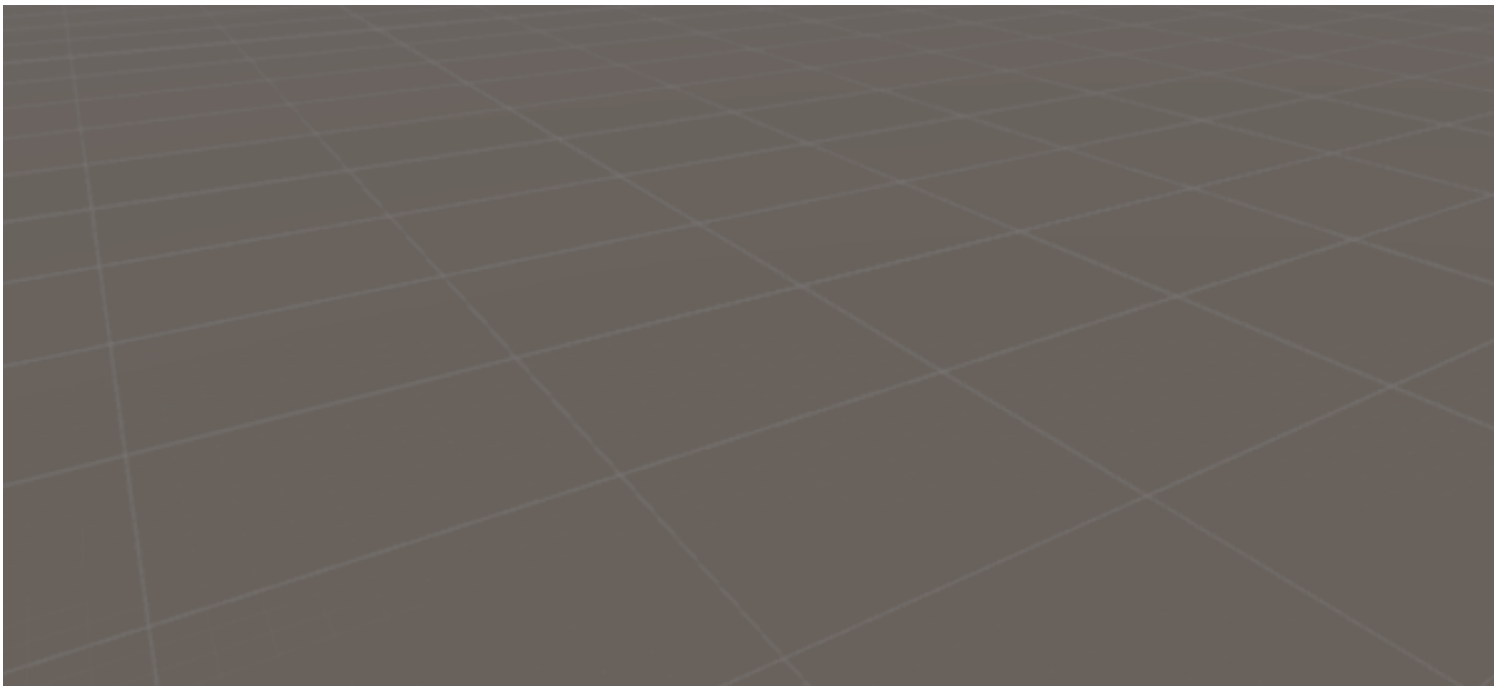
`Vector3.zero` and `Vector3.one` are typically used in connection to a *normalised* `Vector3`; that is, a `Vector3` where the `x`, `y` and `z` values have a magnitude of 1. As such, `Vector3.zero` represents the lowest value, whilst `Vector3.one` represents the largest value.

`Vector3.zero` is also commonly used to set the default position on object transforms.

The following class uses `Vector3.zero` and `Vector3.one` to inflate and deflate a sphere.

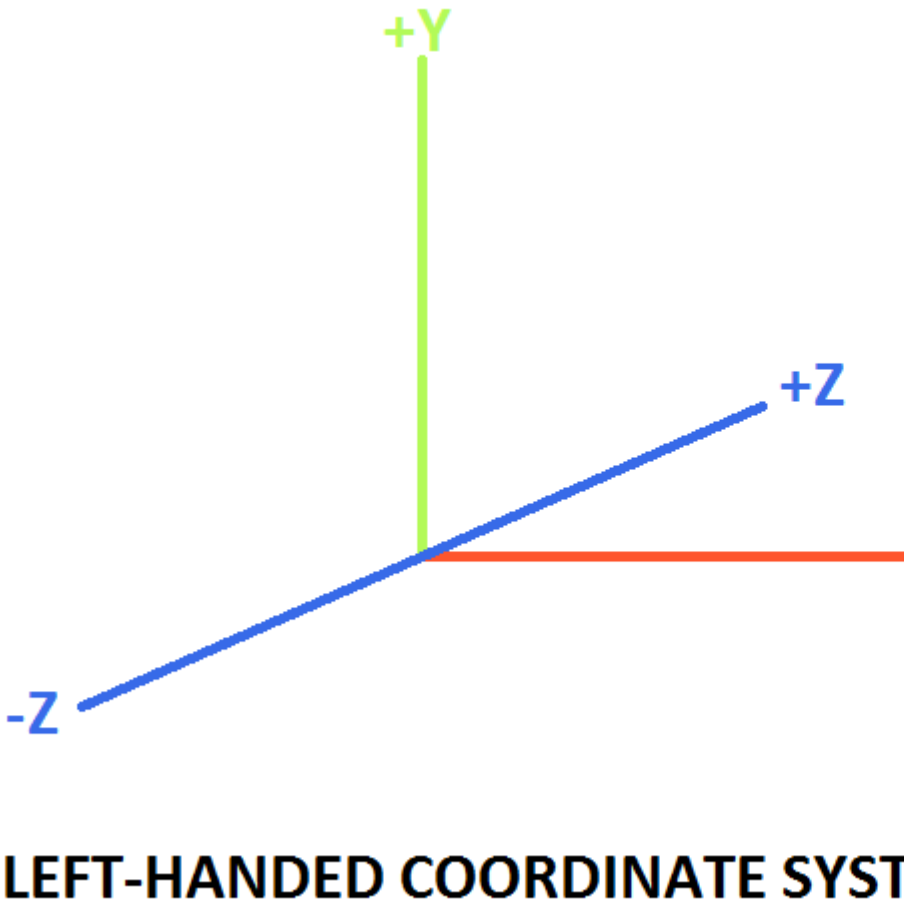
```
using UnityEngine;

public class Inflater : MonoBehaviour
{
    <summary>A sphere set up to inflate and deflate between two values.</summary>
```

```
    public ScaleBetween sphere;

    ///<summary>On start, set the sphere GameObject up to inflate
    /// and deflate to the corresponding values.</summary>
    void Start()
    {
        // Vector3.zero = Vector3(0, 0, 0); Vector3.one = Vector3(1, 1, 1);
        sphere.SetScale(Vector3.zero, Vector3.one);
    }
}
```



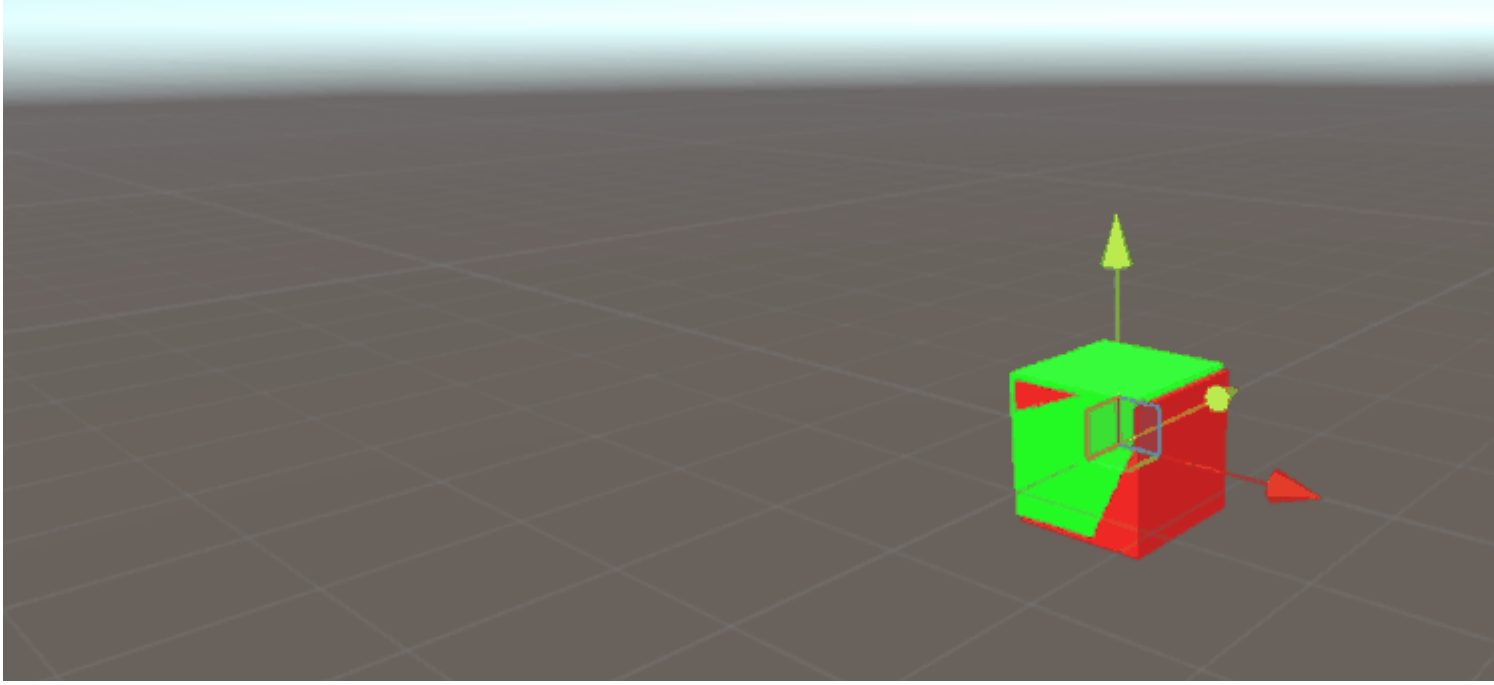
Static Directions

The static directions can be useful in a number of applications, with direction along the positive and negative of all three axis. It is important to note that Unity employs a left-handed coordinate system, which has an affect on direction.



The following class uses the static `Vector3` directions to move objects along the three axis.

```
using UnityEngine;

public class StaticMover : MonoBehaviour
{
    <summary>GameObjects set up to move back and forth between two directions.</summary>
    public MoveBetween xMovement, yMovement, zMovement;

    ///<summary>On start, set each MoveBetween GameObject up to move
    /// in the corresponding direction(s).</summary>
    void Start()
    {
        // Vector3.left = Vector3(-1, 0, 0); Vector3.right = Vector3(1, 0, 0);
        xMovement.SetDirections(Vector3.left, Vector3.right);

        // Vector3.down = Vector3(0, -1, 0); Vector3.up = Vector3(0, 0, 1);
        yMovement.SetDirections(Vector3.down, Vector3.up);

        // Vector3.back = Vector3(0, 0, -1); Vector3.forward = Vector3(0, 0, 1);
        zMovement.SetDirections(Vector3.back, Vector3.forward);
    }
}
```



Index

Value	x	y	z	Equivalent `new Vector3()` method
`Vector3.zero`	0	0	0	`new Vector3(0, 0, 0)`
`Vector3.one`	1	1	1	`new Vector3(1, 1, 1)`
`Vector3.left`	-1	0	0	`new Vector3(-1, 0, 0)`
`Vector3.right`	1	0	0	`new Vector3(1, 0, 0)`
`Vector3.down`	0	-1	0	`new Vector3(0, -1, 0)`
`Vector3.up`	0	1	0	`new Vector3(0, 1, 0)`
`Vector3.back`	0	0	-1	`new Vector3(0, 0, -1)`
`Vector3.forward`	0	0	1	`new Vector3(0, 0, 1)`

Creating a Vector3

A `Vector3` structure can be created in several ways. `Vector3` is a struct, and as such, will typically need to be instantiated before use.

Constructors

There are three built in constructors for instantiating a `Vector3`.

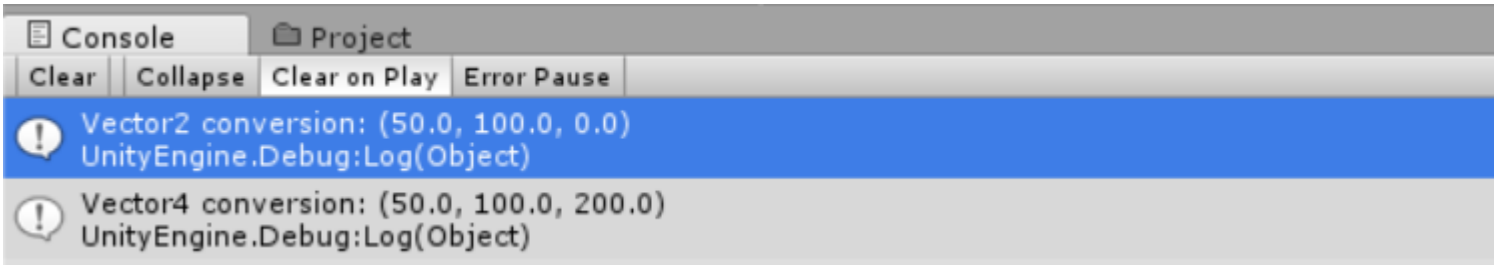
Constructor	Result
`new Vector3()`	Creates a `Vector3` structure with co-ordinates of (0, 0, 0).
`new Vector3(float x, float y)`	Creates a `Vector3` structure with the given `x` and `y` co-ordinates. `z` will be set to 0.
`new Vector3(float x, float y, float z)`	Creates a `Vector3` structure with the given `x`, `y` and `z` co-ordinates.

Converting from a `Vector2` or `Vector4`

While rare, you may run into situations where you would need to treat the co-ordinates of a `Vector2` or `Vector4` structure as a `Vector3`. In such cases, you can simply pass the `Vector2` or `Vector4` directly into the `Vector3`, without previously instantiating it. As should be assumed, a `Vector2` struct will only pass `x` and `y` values, while a `Vector4` class will omit its `w`.

We can see direct conversion in the below script.

```
void VectorConversionTest()
{
    Vector2 vector2 = new Vector2(50, 100);
    Vector4 vector4 = new Vector4(50, 100, 200, 400);

    Vector3 fromVector2 = vector2;
    Vector3 fromVector4 = vector4;

    Debug.Log("Vector2 conversion: " + fromVector2);
    Debug.Log("Vector4 conversion: " + fromVector4);
}
```



Applying Movement

The `Vector3` structure contains some static functions that can provide utility when we wish to apply movement to the `Vector3`.

Lerp

and `LerpUnclamped`

The lerp functions provide movement between two co-ordinates based off a provided fraction. Where `Lerp` will only permit movement between the two co-ordinates, `LerpUnclamped` allows for fractions that move outside of the boundaries between the two co-ordinates.

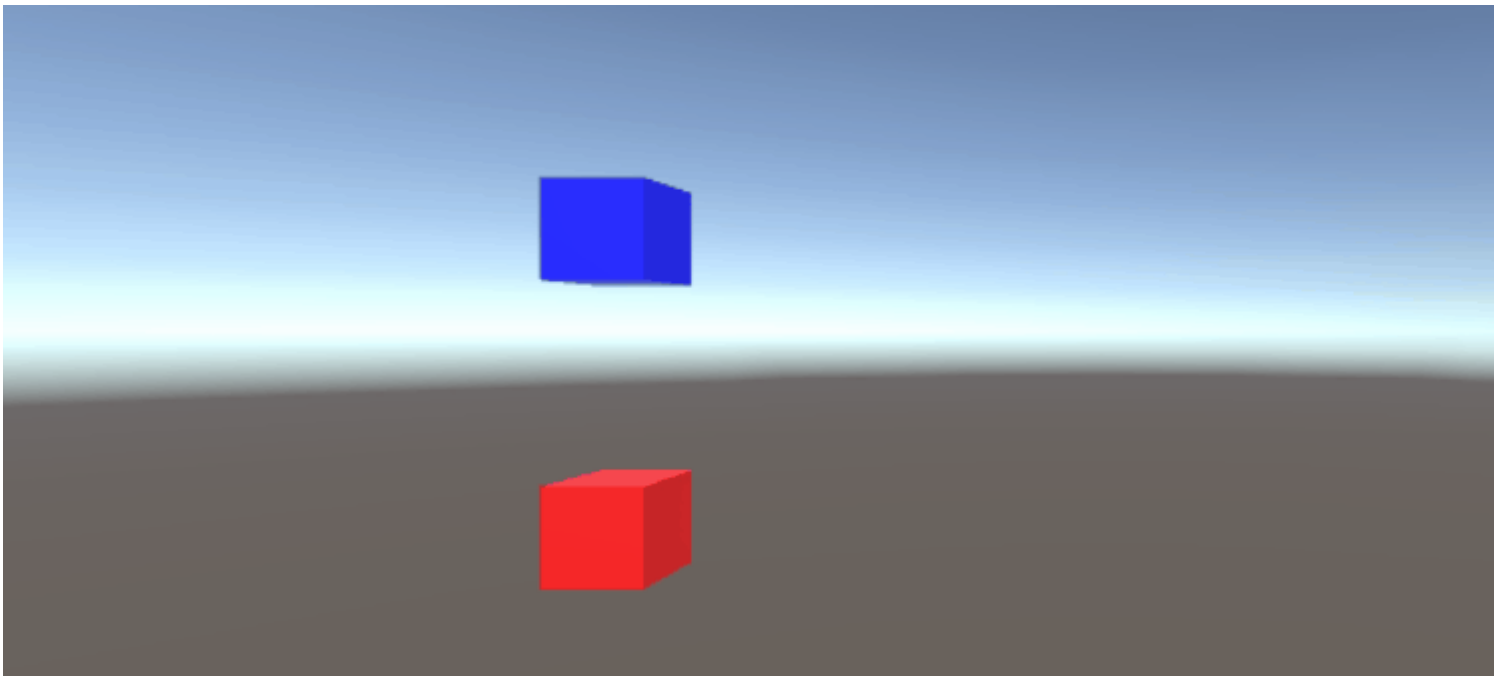
We provide the fraction of movement as a `float`. With a value of `0.5`, we find the midpoint between the two `Vector3` co-ordinates. A value of `0` or `1` will return the first or second `Vector3`, respectivley, as these values either correlate to no movement (thus returning the first `Vector3`), or completed movement (this returning the second `Vector3`). It is important to note that neither function will accommodate for change in the movement fraction. This is something we need to manually account for.

With `Lerp`, all values are clamped between `0` and `1`. This is useful when we want to provide movement towards a direction, and do not want to overshoot the destination. `LerpUnclamped` can take any value, and can be used to provide movement *away* from the destination, or *past* the destination.

The following script uses `Lerp` and `LerpUnclamped` to move an object at a consistent pace.

```
using UnityEngine;

public class Lerping : MonoBehaviour
{
    /// <summary>The red box will use Lerp to move. We will link
    /// this object in via the inspector.</summary>
    public GameObject lerpObject;
    /// <summary>The starting position for our red box.</summary>
    public Vector3 lerpStart = new Vector3(0, 0, 0);
    /// <summary>The end position for our red box.</summary>
    public Vector3 lerpTarget = new Vector3(5, 0, 0);

    /// <summary>The blue box will use LerpUnclamped to move. We will
    /// link this object in via the inspector.</summary>
    public GameObject lerpUnclampedObject;
    /// <summary>The starting position for our blue box.</summary>
    public Vector3 lerpUnclampedStart = new Vector3(0, 3, 0);
    /// <summary>The end position for our blue box.</summary>
    public Vector3 lerpUnclampedTarget = new Vector3(5, 3, 0);

    /// <summary>The current fraction to increment our lerp functions by.</summary>
    public float lerpFraction = 0;

    private void Update()
    {
        // First, I increment the lerp fraction.
        // delaTime * 0.25 should give me a value of +1 every second.
        lerpFraction += (Time.deltaTime * 0.25f);

        // Next, we apply the new lerp values to the target transform position.
        lerpObject.transform.position
            = Vector3.Lerp(lerpStart, lerpTarget, lerpFraction);
        lerpUnclampedObject.transform.position
            = Vector3.LerpUnclamped(lerpUnclampedStart, lerpUnclampedTarget, lerpFraction);
```

```
    }
}
```



MoveTowards

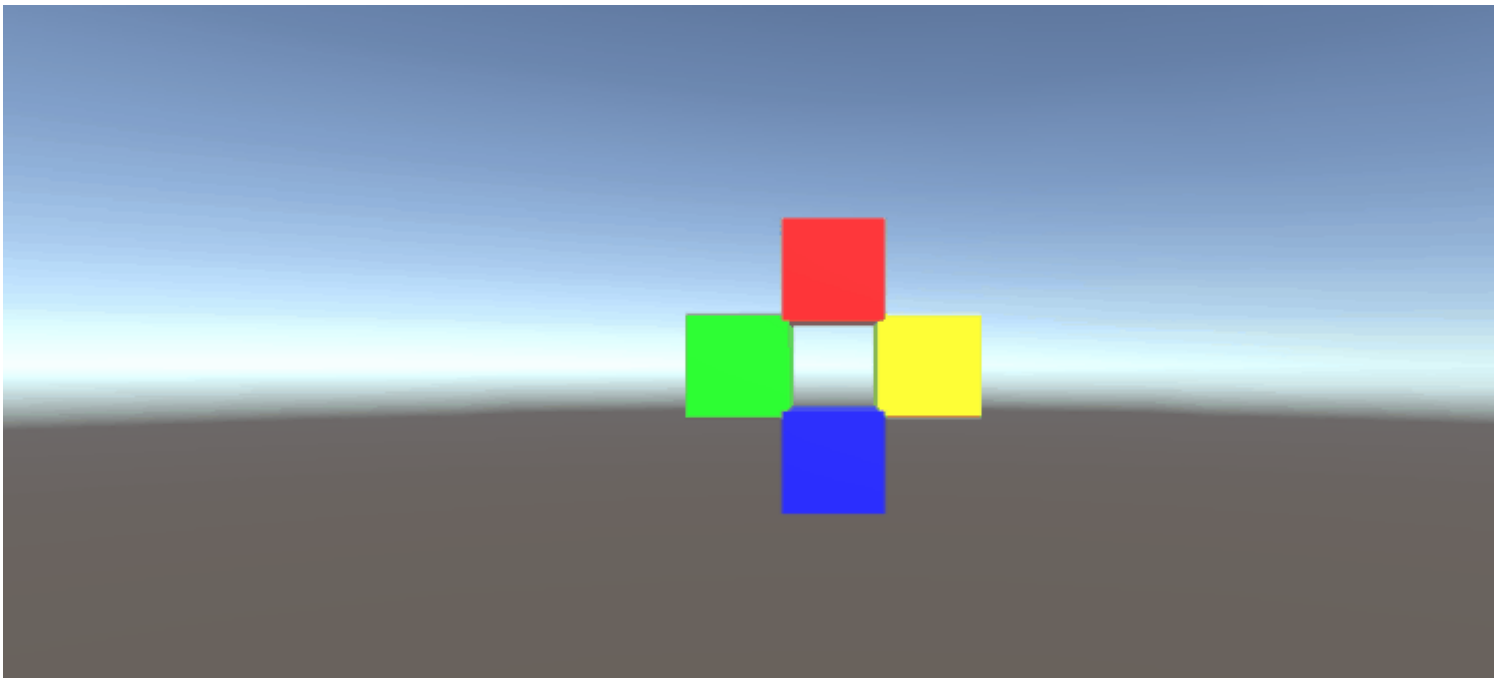
`MoveTowards` behaves *very similar* to `Lerp`; the core difference is that we provide an actual *distance* to move, instead of a *fraction* between two points. It is important to note that `MoveTowards` will not extend past the target `Vector3`.

Much like with `LerpUnclamped`, we can provide a *negative* distance value to move *away* from the target `Vector3`. In such cases, we never move past the target `Vector3`, and thus movement is indefinite. In these cases, we can treat the target `Vector3` as an "opposite direction"; as long as the `Vector3` points in the same direction, relative to the start `Vector3`, negative movement should behave as normal.

The following script uses `MoveTowards` to move a group of objects towards a set of positions using a smoothed distance.

```
using UnityEngine;

public class MoveTowardsExample : MonoBehaviour
{
    /// <summary>The red cube will move up, the blue cube will move down,
    /// the green cube will move left and the yellow cube will move right.
    /// These objects will be linked via the inspector.</summary>
    public GameObject upCube, downCube, leftCube, rightCube;
    /// <summary>The cubes should move at 1 unit per second.</summary>
    float speed = 1f;

    void Update()
    {
```

```
        // We determine our distance by applying a deltaTime scale to our speed.
        float distance = speed * Time.deltaTime;

        // The up cube will move upwards, until it reaches the
        //position of (Vector3.up * 2), or (0, 2, 0).
        upCube.transform.position
            = Vector3.MoveTowards(upCube.transform.position, (Vector3.up * 2f), distance);

        // The down cube will move downwards, as it enforces a negative distance..
        downCube.transform.position
            = Vector3.MoveTowards(downCube.transform.position, Vector3.up * 2f, -distance);

        // The right cube will move to the right, indefinetly, as it is constantly updating
        // its target position with a direction based off the current position.
        rightCube.transform.position = Vector3.MoveTowards(rightCube.transform.position,
            rightCube.transform.position + Vector3.right, distance);

        // The left cube does not need to account for updating its target position,
        // as it is moving away from the target position, and will never reach it.
        leftCube.transform.position
            = Vector3.MoveTowards(leftCube.transform.position, Vector3.right, -distance);
    }
}
```



SmoothDamp

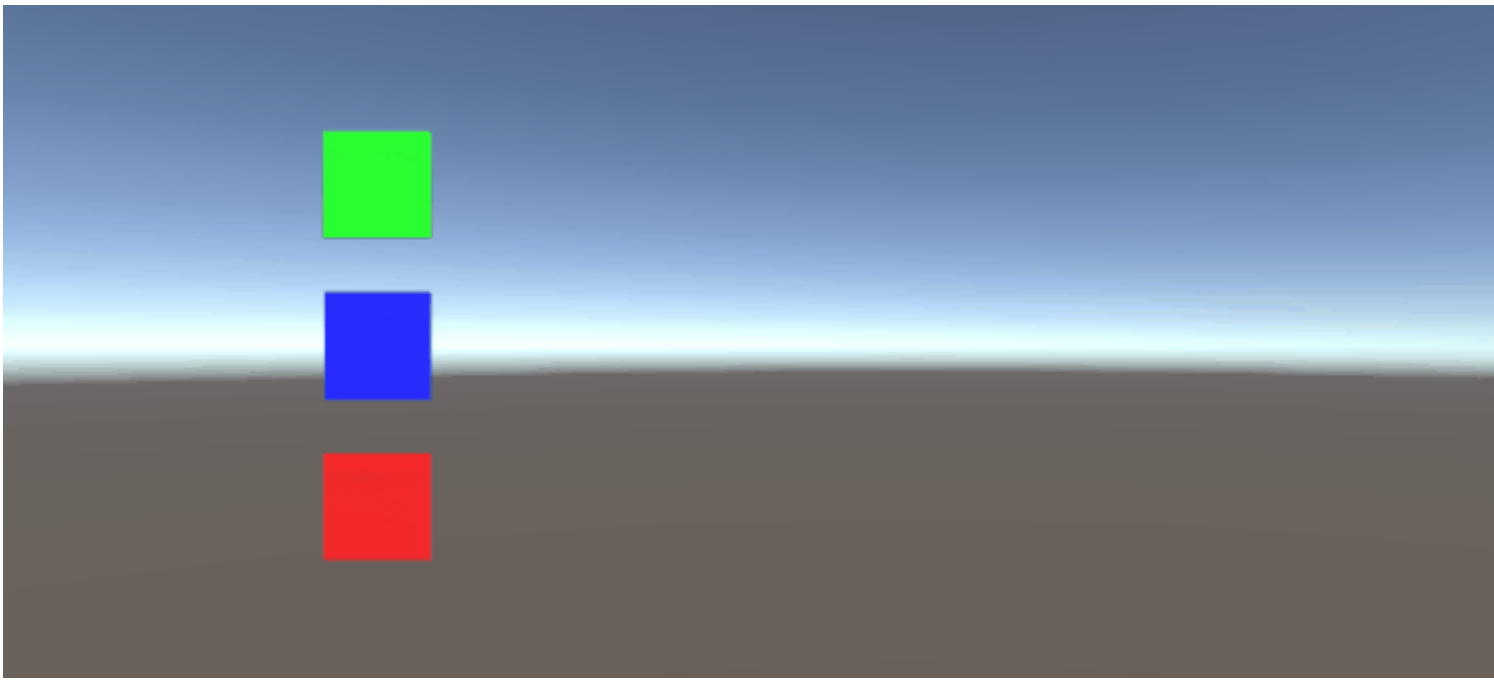
Think of `SmoothDamp` as a variant of `MoveTowards` with built in smoothing. According to official documentation, this function is most commonly used to perform smooth camera following.

Along with the start and target `Vector3` coordinates, we must also provide a `Vector3` to represent the velocity, and a `float` representing the *approximate* time it should take to complete the movement. Unlike previous examples, we provide the velocity as a *reference*, to be incremented, internally. It is important to take note of this, as changing velocity outside of the function while we are still performing the function can have undesired results.

In addition to the *required* variables, we may also provide a `float` to represent the maximum speed of our object, and a `float` to represent the time gap since the previous `SmoothDamp` call to the object. We do not *need* to provide these values; by default, there will be no maximum speed, and the time gap will be interpreted as `Time.deltaTime`. More importantly, if you are calling the function one per object inside a `MonoBehaviour.Update()` function, you should *not* need to declare a time gap.

```
using UnityEngine;

public class SmoothDampMovement : MonoBehaviour
{
    /// <summary>The red cube will imitate the default SmoothDamp function.
    /// The blue cube will move faster by manipulating the "time gap", while
    /// the green cube will have an enforced maximum speed. Note that these
    /// objects have been linked via the inspector.</summary>
    public GameObject smoothObject, fastSmoothObject, cappedSmoothObject;

    /// <summary>We must instantiate the velocities, externally, so they may
    /// be manipulated from within the function. Note that by making these
    /// vectors public, they will be automatically instantiated as Vector3.Zero
    /// through the inspector. This also allows us to view the velocities,
    /// from the inspector, to observe how they change.</summary>
    public Vector3 regularVelocity, fastVelocity, cappedVelocity;

    /// <summary>Each object should move 10 units along the X-axis.</summary>
    Vector3 regularTarget = new Vector3(10f, 0f);
    Vector3 fastTarget = new Vector3(10f, 1.5f);
    Vector3 cappedTarget = new Vector3(10f, 3f);

    /// <summary>We will give a target time of 5 seconds.</summary>
    float targetTime = 5f;

    void Update()
    {
        // The default SmoothDamp function will give us a general smooth movement.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime);

        // Note that a "maxSpeed" outside of reasonable limitations should not have any
        // effect, while providing a "deltaTime" of 0 tells the function that no time has
        // passed since the last SmoothDamp call, resulting in no movement, the second time.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime, 10f, 0f);

        // Note that "deltaTime" defaults to Time.deltaTime due to an assumption that this
        // function will be called once per update function. We can call the function
        // multiple times during an update function, but the function will assume that enough
        // time has passed to continue the same approximate movement. As a result,
        // this object should reach the target, quicker.
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);

        // Lastly, note that a "maxSpeed" becomes irrelevant, if the object does not
        // realistically reach such speeds. Linear speed can be determined as
        // (Distance / Time), but given the simple fact that we start and end slow, we can
        // infer that speed will actually be higher, during the middle. As such, we can
        // infer that a value of (Distance / Time) or (10/5) will affect the
```

```
        // function. We will half the "maxSpeed", again, to make it more noticeable.
        cappedSmoothObject.transform.position = Vector3.SmoothDamp(
            cappedSmoothObject.transform.position,
            cappedTarget, ref cappedVelocity, targetTime, 1f);
    }
}
```



Read Vector3 online: https://riptutorial.com/unity3d/topic/7827/vector3

Chapter 40: Virtual Reality (VR)

Examples

VR Platforms

There are two main platforms in VR, one is mobile platform, like **Google Cardboard**, **Samsung GearVR**, the other is PC platform, like **HTC Vive, Oculus, PS VR**...

Unity officially supports the **Oculus Rift**, **Google Carboard**, **Steam VR**, **Playstation VR**, **Gear VR** , and the **Microsoft Hololens**.

Most platforms have their own support and sdk. Usually, you need to download the sdk as an extension firstly for unity.

SDKs:

- Google Cardboard
- Daydream Platform
- Samsung GearVR (integrated since Unity 5.3)
- Oculus Rift
- HTC Vive/Open VR
- Microsoft Hololens

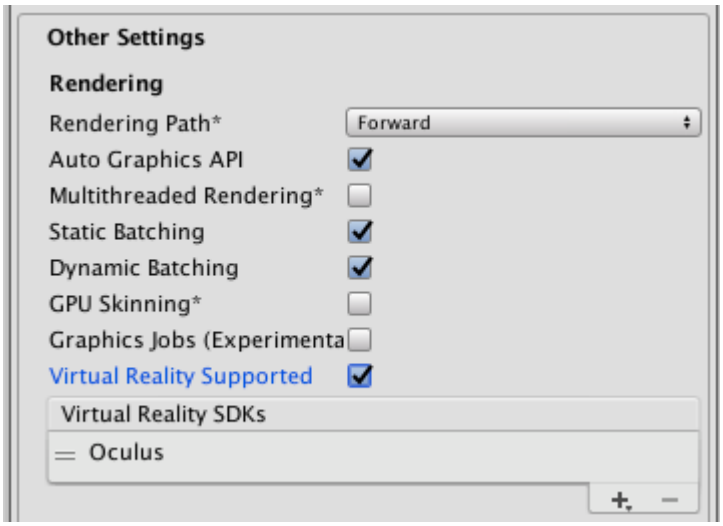
Documentation:

- Google Cardboard/Daydream
- Samsung GearVR
- Oculus Rift
- HTC Vive
- Microsoft Hololens

Enabling VR support

In Unity Editor, open **Player Settings** (Edit > Project Settings > Player).

Under **Other Settings**, check *Virtual Reality Supported*.



Add or remove VR devices for each build target in the *Virtual Reality SDKs* list under the checkbox.

Hardware

There is a necessary hardware dependency for a VR application, that usually depends on the platform that you're building for. There are 2 broad categories for hardware devices based on their motion capabilities:

1. 3 DOF (Degrees of Freedom)
2. 6 DOF (Degrees of Freedom)

3 DOF means that the motion of the Head-Mounted Display (HMD) is constrained to operate in 3 dimensions that is rotate about the three orthogonal axes centered on the HMDs center of gravity - the longitudinal, vertical and horizontal axes. Motion about the longitudinal axis is called roll, motion about the lateral axis is called pitch and motion about the perpendicular axis is called yaw, similar principles that govern motion of any moving object like an aeroplane or a car, which means that although you will be able to see in all X, Y, Z directions by the motion of your HMD in the Virtual environment, but you wouldn’t be able to move or touch anything (motion by an additional bluetooth controller is not the same).

However, 6 DOF allows for a room-scale experience wherein you can also move about the X,Y and Z axis apart from the roll, pitch and yaw motions about its centre of gravity, hence the 6 degree of freedom.

Currently a Room-scale VR facilitated for 6 DOF requires high computation performance with a high-end graphic card and RAM that you probably won’t get from your standard laptops and will require a desktop computer with optimal performance and also at least 6ft × 6ft free space, whereas a 3 DOF experience can be achieved by just a standard smart phone with an inbuilt gyro (which is inbuilt in most modern smart phones that cost about $200 or more).

Some common devices available in the market today are:

- Oculus Rift (6 DOF)
- HTC Vive (6 DOF)

- Daydream (3 DOF)
- Gear VR Powered by Oculus (3 DOF)
- Google Cardboard (3 DOF)

Read Virtual Reality (VR) online: https://riptutorial.com/unity3d/topic/5787/virtual-reality--vr-

Credits

S. No	Chapters	Contributors
1	Getting started with unity3d	Alexey Shimansky, Chris McFarland, Community, Desutoroiya, driconmax, Ḟĭáḿíṅḡ ỏḿƀíé, James Radvan, josephsw, Linus Juhlin, Luís Fonseca, Maarten Bicknese, martinhodler, matiaslauriti, Mike B, Minzkraut, PlanetVaster, R.K123, S. Tarık Çetin, Skyblade, SourabhV, SP., tenpn, tim, user3071284
2	Ads integration	ɬolɐəz əuɬ qoq
3	Android Plugins 101 - An Introduction	Venkat at Axiom Studios
4	Asset Store	JakeD, Trent, zwcloud
5	Attributes	4444, Thundernerd
6	Audio System	R4mbi, ɬolɐəz əuɬ qoq
7	Collision	Ḟĭáḿíṅḡ ỏḿƀíé, jjhavokk, Xander Luciano
8	Communication with server	David Martinez, devon t, Ḟĭáḿíṅḡ ỏḿƀíé, Maxim Kamalov, tim
9	Coroutines	agiro, Fattie, Fehr, Giuseppe De Francesco, Problematic, Skyblade, Thulani Chivandikwa, Thundernerd, ɬolɐəz əuɬ qoq, volvis
10	CullingGroup API	volvis
11	Design Patterns	Ian Newland
12	Extending the Editor	Pierrick Bignet, Skyblade, Thundernerd, ɬolɐəz əuɬ qoq, volvis
13	Finding and collecting GameObjects	Pierrick Bignet, S. Tarık Çetin, volvis
14	How to use asset packages	Ḟĭáḿíṅḡ ỏḿƀíé
15	Immediate Mode Graphical User Interface System	Skyblade, Soaring Code

	(IMGUI)	
16	Importers and (Post)Processors	gman, Skyblade, volvis
17	Input System	Programmer, Skyblade, ɬolɐəz əɥɬ qoq
18	Layers	Arijoon, dreadnought, Light Drake, RamenChef, Skyblade
19	Mobile platforms	Airwarfare, Skyblade
20	MonoBehaviour class implementation	matiaslauriti, Skyblade, Thundernerd, user3797758
21	Multiplatform development	user3797758, volvis
22	Networking	David Martinez, driconmax, Rafiwui, RamenChef
23	Object Pooling	Chris McFarland, Ed Marty, lase, matiaslauriti, S. Tarık Çetin, Thulani Chivandikwa, Thundernerd, ɬolɐəz əɥɬ qoq, volvis
24	Optimization	Ed Marty, EvilTak, Ḟ̣ámíṅḡ ỏḿƀié, Grigory, JohnTube, Skyblade, Thulani Chivandikwa, volvis
25	Physics	eunoia, Ḟ̣ámíṅḡ ỏḿƀié, jack jay
26	Prefabs	Brandon Mintern, Dávid Florek, Ḟ̣ámíṅḡ ỏḿƀié, gman, Gnemlock, Guglie, James Radvan, Jean Vitor, josephsw, Lich, matiaslauriti, Skyblade, Thulani Chivandikwa, ɬolɐəz əɥɬ qoq, Woltus, yummypasta
27	Quaternions	matiaslauriti, Tiziano Coroneo, Xander Luciano, yummypasta
28	Raycast	driconmax, Meinkraft, Skyblade, user3570542, volvis, wouterrobot
29	Resources	glaubergft, MadJlzz, Skyblade, Venkat at Axiom Studios
30	ScriptableObject	volvis
31	Singletons in Unity	David Darias, Fehr, James Radvan, JohnTube, matiaslauriti, Maxim Kamalov, Simon Heinen, SP., Tiziano Coroneo, Umair M , volvis, Zze,
32	Tags	Arijoon, Augure, glaubergft, Gnemlock, MadJlzz, Skyblade, Trent
33	Transforms	ADB, Jean Vitor, matiaslauriti, S. Tarık Çetin, Skyblade, Thundernerd, Xander Luciano

34	Unity Animation	4444, Fiery Raccoon, Guglie
35	Unity Lighting	Ḟ̄ámíṅḡ ỏḿƀié
36	Unity Profiler	Amitayu Chakraborty, ForceMagic, RamenChef, Skyblade
37	User Interface System (UI)	Hellium, matiaslauriti, Maxim Kamalov, Programmer, RamenChef, Skyblade, Umair M
38	Using Git source control with Unity	Commodore Yournero, Hacky, James Radvan, matiaslauriti, Max Yankov, Maxim Kamalov, Pierrick Bignet, Ricardo Amores, S. Tarık Çetin, S.Richmond, Skyblade, Thulani Chivandikwa, YsenGrimm, yummypasta
39	Vector3	driconmax, Ḟ̄ámíṅḡ ỏḿƀié, Gnemlock
40	Virtual Reality (VR)	4444, Airwarfare, Guglie, pew., Pratham Sehgal, tim