Ben Yep

V 1.2

A Lesson in C# and Unity (general dev)

**Notice: I encountered an issue where very large ScrollRects were killing performance. At the time, development using Unity became impractical for my purposes. I am not sure if a work-around has been found. However, this guide may still be helpful for programmers looking to use Unity to develop games.**

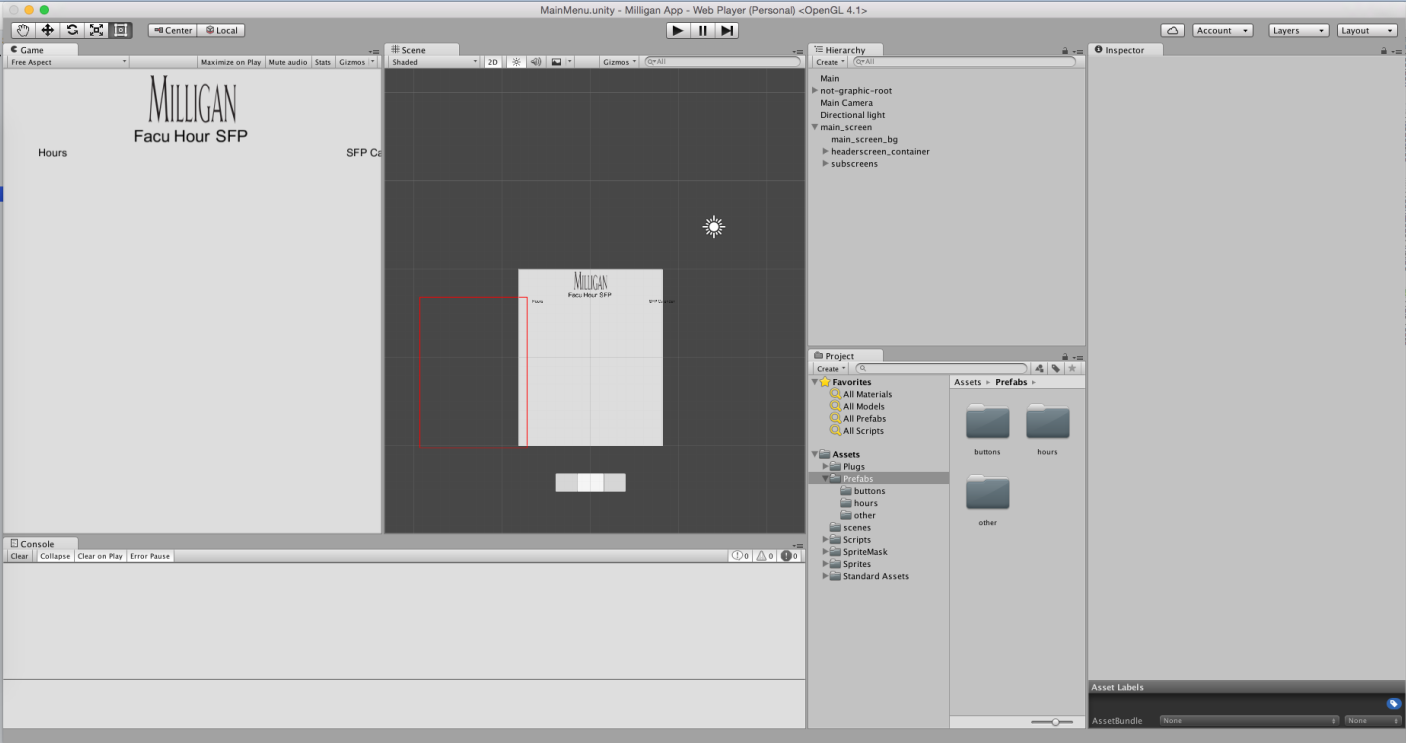
This guide assumes a moderate knowledge of Object Oriented Programming, and a slight familiarity with C#.

Introduction

I wrote this lesson as a guide to help accelerate your familiarity with Unity and C#. Unity, while immensely useful, is tedious to become acquainted with.   
  
As some of you may know I am developing the new Milligan App right now. It has been my sandbox for learning Unity and it has certainly been an experience. I am currently in the process of modernizing the app, which involved re-creating its user interface. This is the particularly annoying part. Fortunately, you don’t have to worry about setting this up because I have attached included in this distribution a scene with all necessary dependencies for this lesson. **Please create a new Unity Project and import import\_me.unitypackage from “import package->custom package.”** Most of this guide will use this scene to explain things, so it is important that you import the package and have it open to get the most from this guide.

The Basics

The Editor



This is the layout that I use for my Unity development. I will assume that you are also using this layout.

**The editor** is everything you see here BESIDES the “game” tab on the left. That is usually referred to as “run-time,” because it is the app as it will be seen by the user.

I won’t be lengthy, most of it is the same as your standard IDE so I will not explain those parts.

**Play Mode**

Unity is unlike many IDEs because it allows interactive GUI testing in the editor while the game is being played. In other words, if you press the play button (CTRL+P OR COMMAND+P), the scene on the left becomes active. This tab is the game from as if it the user. It shows what the user will see when they are using your app. **However**, the tab on the right (“scene”) is still usable. It allows you to manually manipulate objects during play mode. Likewise, you can make changes in the hierarchy and inspector on the far right as well. **Any changes you make in play mode are not reflected once play mode is exited.** This is enormously useful. You can do extreme editing without worrying about permanently changing anything. Use this to your advantage.

Also, you should practice, navigating the editor in 3D mode which you can toggle in the top of the scene tab. This is useful even in 2D apps when dealing with issues that involve things improperly layering.

**Hierarchy & Assets & Inspector**

Pretty self-explanatory. Keep your assets organized. Keep your scripts in one folder, sprites in another, and prefabs (I will explain what those are later) in another.

**Assets in the Example**

Plugs: Contains HTMLAgilityPack, a 3rd party tool that I included in this scene. I downloaded it from the internet. It is used to parse html.

Prefabs: Contains all prefabs that I use in this example (read on).

Scenes: Contains all scenes. I believe I only included one or two scenes. The only one that needs to be included in the build is MainMenu.

Scripts: Contains ALL scripts written by myself (and a couple snippets from the internet). There is no code I have written outside of this folder.

Basic Unity Organization

**Scene**

Unity is originally built to make games. Scenes are like levels. If you load a new scene, you completely unload everything else. So actually scenes are quite limiting if you are looking to create something modern, dynamic, and oh so beautiful. When you create a new project you are already in a scene, the default scene. You might make an additional scene if you had to, for example, create a loading screen or if you for some reason need to completely shift gears in your app but for the most part you will likely only use one scene.

**Prefabs**

Prefabs are weird. See, in Unity, graphics are not organized solely by classes. In fact, they are not necessarily tied up with any code at all by default. But in order to treat a graphical object like a class, it must be a prefab.

Prefabs are simply templates of graphical objects. They can be reused and easily edited. When the prefab is edited, all instances of the prefab are affected. If an instance of a prefab is edited significantly in the editor, that prefab instance becomes **broken**. It will not sync up all instances unless the prefab is directly edited in the library (using the inspector), or an instance is edited and then the “apply” button is selected on the top of the inspector.  
TO CREATE A PREFAB:  
Simple. Just drag any object from the hierarchy into your assets. Anywhere (I recommend organized in a prefabs folder). Done. Name it what you want but I recommend naming it using CaptialCamelCase like most classes are named. It’s convention, as it may be directly linked to a class title.

TO INSTANTIATE A PREFAB:  
Simply drag the object onto the screen from your prefab folder in assets.

TO EDIT A PREFAB:

You will notice that at the top of all objects that are (or were at one point) a prefab, you will see an apply button. When you make a change to an object in blue in the hierarchy (a prefab instance), it will turn black indicating that it is now a broken prefab. To turn this into a prefab again by **editing the prefab itself and every instance of the prefab,** click apply. **Be careful**. If you want to disassociate an object from a prefab, you **cannot**. So in some situations clicking “apply” might cause changes that you did not mean to incur. Keep track of which of your objects are connected to prefabs, and use the apply button intelligently.

**Components**

All gameObjects (basically, just any Unity object), are comprised of components. An object’s components can be viewed in the inspector. You can obtain components in code through using the method GetComponent<ComponentType>(). The components manage the behavior and properties of all objects. Remember that Unity is as much component-oriented as it is object oriented. All objects that extend MonoBehavior (that is the base class for all unity “gameObjects” have components, and these components define almost everything that the object is. **Check out tricks and tips: Editor (at the end of this document) right now to see an easy way to avoid hours of struggles with components.**

Let’s go over some basic components:

*Transform* – Included in all components. You can’t remove it. It can be replaced by a RectTransform and **usually** is. For example, if you add an image component to the object the transform component will automatically be converted to a RectTransform component.

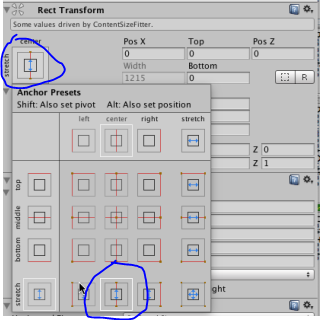
*Image* – This one is easy. Use it to add an image (or Sprite) from your assets to an object. **Note that an image component will not render unless its object is a child of a Canvas (it can be a grandchild, etc.). If you want to experiment, first add an object with a Canvas component to the hierarchy and then add the image component to a child of that canvas.**  
Image components automatically add a “canvas renderer” component to the object as well. This is a dependency. It does not contain any variables accessible in the editor, but it does contain important functions that can be accessed through code.

*Layout Element* – Use this component along with *Vertical Layout Group*, *Horizontal Layout Group*, *Grid Layout Group*, and *Content Size Fitter* to create html-like organization of elements. Enormously useful for creating lists of objects. Look at how I have used combinations of Layout Element, Vertical/Horizontal Layout Group, and Content Size Fitter in the “subscreens” object in the hierarchy. Note that preferred width/height is in most situations the equivalent of max-width/height. Note that these values are useful for forcing width and height values programmatically, unlike width and height which can sometimes be locked to editing.

*Text, Image,* and *ScrollRect* and *Scrollbar* are all other common components. Try experimenting with different components to see how they work.

*Script* – Attach a script to an object. **When a script is attached to an object that is an instance of a prefab, and if the class-name is identical to the prefab name, that object is treated like an instance of the class.** In this example, FacultyMember/FacultyMember.cs is a good example of this.

**Anchors**

****

Anchors are amazing. Anchors are a part of the RectTransform component and are used to solve the issue of things appearing radically differently (almost always breaking) due to even slight changes in aspect ratio or resolution. Remember that not all phones have the same size screen. Anchors can be positioned manually, but more often than not you will use the anchor presets. Simply click the anchor box of the RectTransform to select a preset.

Anchors define how an object is positioned. If all anchors are kept together, than the point that they meet is treated as local position 0,0,0 compared to its parent. You can pick where the anchor lies, be it left top, center top, bottom left, etc. You can also choose to stretch one or both axes (plural of axis, not like the thing Gimli carries) to the lengths of its parent object. While leaving zero or one axes or freedom respectively with 0,0,0 being set wherever you choose. Experiment with this and see how I use anchors. These can be accessed programmatically through accessing the component RectTransform.

**Scripts**

In Unity, all of your code will be in scripts. Period. Most of it will be in a script attached to a “Main” object that will arbitrarily be your script to perform most of your operations. The main coding file. But we won’t get into that quite yet. First, you must understand how classes work in Unity.  
**One again, classes are tied to graphical objects in unity when a script is attached to an object via a script component. The Class name and prefab name must all match.**  
However, it isn’t essential to do this. It is also possible to just attach a script to any object and the “Awake” and “Start” methods will be called just the same but the object will NOT be treated as the child of the class if the class does not share the name of the prefab.

Things every Scene should have

Every scene needs to have some certain objects in it. Here they are:

**Main and <Scene>Main** – See basic scripting – organization

**Directional light ­**– Why do my images look so dark? I need a light source. Add one.

Main Camera – Orthographic. **Orthographic size of 1.** Why scale the units to any other size than one I am not sure what the point is. If you use sprites often it still makes more sense to adjust the scales of the sprites and not touch the orthographic size. As far as I understand, this is common practice for 2D-apps.

Event System – This manages all of your events like clicks. Have exactly one in every scene. Just add it from the components menu (see tricks and tips).

**Probably a Canvas** – you need a canvas to do anything UI related. It needs to be the parent of all UI. **The canvas should have a plane distance of at least 100. IMO. For 2D apps there is no reason to take advantage of ample amounts of Z-space that I know of.**

Basic Scripting

Organization

**From scene to shining scene**

Every scene should have within it a <SceneName>Main object with a <SceneName>Main script attached to it (where <SceneName> is the name of the current scene. Every scene should also have a “Main” object with a “Main” script attached to it, **however** you should only **add** it to the first scene that loads in your project, as we are going to use a useful trick to force that object not to unload from scene to scene.

It’s a line of code actually:  
DontDestroyOnLoad(this);  
Place that in the “Awake” method of your Main script, first line. There you go, you just configured the file where most of your code will go.

**A typical .CS (C Sharp file)**

using UnityEngine;

using etc blah blah blah;

public class ClassName : MonoBehavior { //Or some class that extends //MonoBehavior

public variable blah;

private variable blah=foobar;

etc.

void Awake(){}//this method is called when the object is constructed in CODE, not graphically. Only handle initializing data here.

Void Start(){}//this one calls when everything is ready.

Void Update(){}//called every frame.

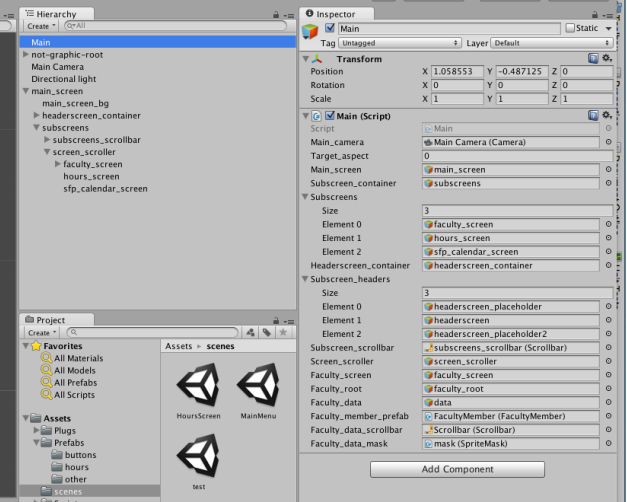
}

Public class SomeOtherClassName{// you can have more classes than one in a //file. If you make them public it doesn’t matter where you put your logical //classes.

//etc.

}

**Assigning public variables from the editor**



Alright this part may seem unnatural to programmers at first but it turns out to be one of the coolest, easiest things. So when you declare a public variable in any script, you can actually choose what the value of that public variable is (unless you specify [HideInInspector]). So, let’s say you have your Main script attached to your main object. You view your main object in the inspector. In the main script you write:

Public Camera main\_camera;

Well, now in the inspector you will SEE that main\_camera and you can drag your actual camera from the hierarchy into that slot to assign that component to that variable. This is the primary reason to turn your main classes into singletons (covered in the next chapter).

Code Tips and Snippets

**Aspect Ratios**  
So here’s something useful. I have found out how to make your UI fit to any aspect ratio and resolution device. It’s actually quite simple.

Set your canvas to be in camera-space, and then set the camera to be your main camera. Grab the “initialize camera” method that I obtained from StackOverflow. You can find it in my Main.cs file. Set what you want your target aspect to be and then call this method in the beginning of your code as I did to adjust the viewport of your camera to your desired aspect ratio.

**Creating Psuedo-Global space (singleton)**

You may have noticed that Unity does not have any sort of global space. This can be an issue when certain aspects of the game need to be accessed globally. Specifically it is an issue with gameObjects because gameObjects cannot be assigned in the editor if they are static. This issue can be solved by turning the main scripts into singletons. You can do this by adding a:  
[HideInInspector] public MonoBehaviour current\_screen;

To your methods in your class. Then, in Awake(), put:

Main.main=this;

There you go. Now you can simply access any variable in your main by referencing Main.main. For example, say an item in a list needs to access the value of its scrollbar. Well, now you can store that scrollbar as a public variable in Main, and access it in the list item’s script by doing something like:

Main.main.scrollbar.value=.5f;

**What are the f’s after some of the numbers? (i.e. 5f)**

That is a fast way to cast to a float since all constant values are integers by default.

Configuring Unity

Building

**Build Settings** in “file” is your best friend. Notice that part at the top which says something like “scenes to include in build.” Include all of the scenes you want. You may be able to build and run but it is possible that you will not be able to. You may have to build the app and manually put it on your device, especially for android apps. You can switch between building platforms in the **build settings** window. It doesn’t particularly matter which build environment you are in for developing since we use code to make aspect ratio pretty arbitrary, and our app will be mostly UI based.

Android Configuration

Go to Unity->Preferences->External tools and you will see that you need to install and download Android SDKs and such. Do that and include them here if you want to build for android.

Misc. Tips & Tricks

Hotkeys

New Scene: CTRL+N

New Blank Object: CTRL+SHIFT+N

Editor

Components:

Right click the hierarchy to see a list of some default objects to construct. These objects come with components already and are very useful for when you are still getting used to components. Heck, even when you are used to it is still useful.

An example of how this is useful: with canvases. Canvases, when added by using this template menu, also contain canvas-scalars and graphic-raycaster components. These are standards to be included with a canvas and they are important, although they are technically not dependencies.

I will conclude by reminding you that if you ever get stuck while developing, approach the issue logically. Identify the most likely issues and check those first. It’s often a component-related problem so identify every component, look for weird scales and values because Unity loves to mess with those. Look at an object’s parents and even its children. Look to the web, look to your friends. And if you get really, really stuck post your question to the Unity forums. They are quite active. Good luck with your programming and if you have any questions related to this guide you can contact me.  
  
~Ben