**Unity3D Best Practices**

These are a set of best practices I’ve found from a number of other sites (some are copied whole), pared down the ones I don’t agree with and added a few of my own. I really need to go back through these, as some of them are no longer appropriate for Unity3D version 5+.

**Process**

**Each team member should have a second copy of the project checked out for testing**if you are using version control. After changes, this second copy, the clean copy, should be updated and tested. No-one should make any changes to their clean copies. This is especially useful to catch missing assets.

**Scene Organization**

**Use named empty game objects as scene folders.**Carefully organize your scenes to make it easy to find objects.

**Put maintenance prefabs and folders (empty game objects) at 0 0 0.**If a transform is not specifically used to position an object, it should be at the origin. That way, there is less danger of running into problems with local and world space, and code is generally simpler.

**Put your world floor at y = 0.** This makes it easier to put objects on the floor, and treat the world as a 2D space (when appropriate) for game logic, AI, and physics.

**Make the game runnable from every scene.** This drastically reduces testing time. To make all scenes runnable you need to do two things:

First, provide a way to mock up any data that is required from previously loaded scenes if it is not available.

Second, spawn objects that must persist between scene loads with the following idiom:

|  |
| --- |
| myObject = FindMyObjectInScene();  if (myObjet == null)  {  myObject = SpawnMyObject();  } |

**Art**

**Put character and standing object pivots at the base, not in the centre.**This makes it easy to put characters and objects on the floor precisely. It also makes it easier to work with 3D as if it is 2D for game logic, AI, and even physics when appropriate.

**Make all meshes face in the same direction (positive or negative z axis).** This applies to meshes such as characters and other objects that have a concept of facing direction. Many algorithms are simplified if everything have the same facing direction.

**Get the scale right from the beginning.** Make art so that they can all be imported at a scale factor of 1, and that their transforms can be scaled 1, 1, 1. Use a reference object (a Unity cube) to make scale comparisons easy. Choose a world to Unity units ratio suitable for your game, and stick to it.

**Make a two-poly plane** to use for GUI components and manually created particles. Make the plane face the positive z-axis for easy billboarding and easy GUI building.

**Make and use test art**

* Squares labelled for skyboxes.
* A grid.
* Various flat colours for shader testing: white, black, 50% grey, red, green, blue, magenta, yellow, cyan.
* Gradients for shader testing: black to white, red to green, red to blue, green to blue.
* Black and white checkerboard.
* Smooth and rugged normal maps.
* A lighting rig (as prefab) for quickly setting up test scenes.

**Prefabs**

**Use prefabs for everything.** The only game objects in your scene that should not be prefabs should be folders. Even unique objects that are *used only once*should be prefabs. This makes it easier to make changes that don’t require the scene to change.

**Use separate prefabs for specialisation; do not specialise instances.**If you have two enemy types, and they only differ by their properties, make separate prefabs for the properties, and link them in. This makes it possible to

* make changes to each type in one place
* make changes without having to change the scene.

If you have too many enemy types, specialisation should still not be made in instances in the editor. One alternative is to do it procedurally, or using a central file / prefab for all enemies. A single drop down could be used to differentiate enemies, or an algorithm based on enemy position or player progress.

**Link prefabs to prefabs; do not link instances to instances.**Links to prefabs are maintained when dropping a prefab into a scene; links to instances are not. Linking to prefabs whenever possible reduces scene setup, and reduce the need to change scenes.

**As far as possible, establish links between instances automatically.** If you need to link instances, establish the links programmatically. For example, the player prefab can register itself with the GameManager when it starts, or the GameManager can find the Player prefab instance when it starts.

**Don’t put meshes at the roots of prefabs if you want to add other scripts.** When you make the prefab from a mesh, first parent the mesh to an empty game object, and make *that* the root. Put scripts on the root, not on the mesh node. That way it is much easier to replace the mesh with another mesh without loosing any values that you set up in the inspector.

**Use linked prefabs as an alternative to nested prefabs.** Unity does not support nested prefabs, and existing third-party solutions can be dangerous when working in a team because the relationship between nested prefabs is not obvious.

**Use safe processes to branch prefabs.** The explanation use the Player prefab as an example.

Make a risky change to the Player prefab is as follows:

1. Duplicate the Player prefab.
2. Rename the duplicate to \_\_Player\_Backup.
3. Make changes to the Player prefab.
4. If everything works, delete \_\_Player\_Backup.

Do not name the duplicate Player\_New, and make changes to it!

Some situations are more complicated. For example, a certain change may involve two people, and following the above process may break the working scene for everyone until person two finished. If it is quick enough, still follow the process above. For changes that take longer, the following process can be followed:

1. Person 1:
   1. Duplicate the Player prefab.
   2. Rename it to \_\_Player\_WithNewFeature or \_\_Player\_ForPerson2.
   3. Make changes on the duplicate, and commit / give to Person 2.
2. Person 2:
   1. Make changes to new prefab.
   2. Duplicate Player prefab, and call it \_\_Player\_Backup.
   3. Drag an instance of \_\_Player\_WithNewFeature into the scene.
   4. Drag the instance onto the original Player prefab.
   5. If everything works, delete \_\_Player\_Backup and \_\_Player\_WithNewFeature.

**Time**

**Maintain your own time class to make pausing easier.** Wrap Time.DeltaTime andTime.TimeSinceLevelLoad to account for pausing and time scale. It requires discipline to use it, but will make things a lot easier, especially when running things of different clocks (such as interface animations and game animations).

**Spawning Objects**

**Don’t let spawned objects clutter your hierarchy when the game runs.** Set their parents to a scene object to make it easier to find stuff when the game is running. You could use a empty game object, or even a singleton with no behaviour to make it easier to access from code.

**Class Design**

**Use singletons for convenience.**The following class will make any class that inherits from it a singleton automatically:

|  |
| --- |
| public class Singleton<T> : MonoBehaviour where T : MonoBehaviour  {  protected static T instance;  /\*\*  Returns the instance of this singleton.  \*/  public static T Instance  {  get  {  if(instance == null)  {  instance = (T) FindObjectOfType(typeof(T));  if (instance == null)  {  Debug.LogError("An instance of " + typeof(T) +  " is needed in the scene, but there is none.");  }  }  return instance;  }  }  } |

Singletons are useful for managers, such as ParticleManager or AudioManager or GUIManager.

* Avoid using singletons for unique instances of prefabs that are not managers (such as the Player). Not adhering to this principle complicates inheritance hierarchies, and makes certain types of changes harder. Rather keep references to these in yourGameManager (or other suitable God class ;-) )
* Define static properties and methods for public variables and methods that are used often from outside the class. This allows you to write GameManager.Player instead ofGameManager.Instance.player.

**For components, never make variables public that should not be tweaked in the inspector.**Otherwise it *will* be tweaked by a designer, especially if it is not clear what it does.  If it seems unavoidable, use [HideInInspector] tag. Don’t set things to public to make them appear in the inspector. Only properties or members that you want to alter from outside your class should be public. Instead use the [SerializeField] (@SerializeField) attribute to make variables show in the inspector. If you have something that needs to be public but not designer editor then use [HideInInspector] (@HideInInspector). On the same note, don’t make things public just for the fun of it. The more public functions and member variables you have in a class, the more stuff shows up in the drop-down list when you try to access it. Keep it clean.

**Separate interface from game logic.**This is essentially the MVC pattern.

*Any input controller should only give commands to the appropriate components to let them know the controller has been invoked.* For example in controller logic, the controller could decide which commands to give based on the player state. But this is bad (for example, it will lead to duplicate logic if more controllers are added). Instead, the Player object should be notified of the intent of moving forward, and then based on the current state (slowed or stunned, for example) set the speed and update the player facing direction. Controllers should only do things that relate to their own state (the controller does not change state if the player changes state; therefore, the controller should not know of the player state at all). Another example is the changing of weapons. The right way to do it is with a method on PlayerSwitchWeapon(Weapon newWeapon), which the GUI can call. The GUI should not manipulate transforms and parents and all that stuff.

*Any interface component should only maintain data and do processing related to it’s own state.* For example, to display a map, the GUI could compute what to display based on the player’s movements. However, this is game state data, and does not belong in the GUI. The GUI should merely display game state data, which should be maintained elsewhere. The map data should be maintained elsewhere (in the GameManager, for example).

*Gameplay objects should know virtually nothing of the GUI.* The one exception is the pause behaviour, which is may be controlled globally through Time.timeScale (which is not a good idea as well… see ). Gameplay objects should know if the game is paused. But that is all. Therefore, no links to GUI components from gameplay objects.

In general, if you delete all the GUI classes, the game should still compile.

You should also be able to re-implement the GUI and input without needing to write any new game logic.

**Separate state and bookkeeping.** Bookkeeping variables are used for speed or convenience, and can be recovered from the state. By separating these, you make it easier to

* save the game state, and
* debug the game state.

One way to do it is to define a SaveData class for each game logic class. The

|  |
| --- |
| [Serializable]  PlayerSaveData  {  public float health; //public for serialisation, not exposed in inspector  }  Player  {  //... bookkeeping variables  //Don’t expose state in inspector. State is not tweakable.  private PlayerSaveData playerSaveData;  } |

**Don’t use strings for anything other than displayed text.** In particular, do not use strings for identifying objects or prefabs etc. One unfortunate exception is animations, which generally are accessed with their string names.

**Avoid using public index-coupled arrays.**For instance, do not define an array of weapons, an array of bullets, and an array of particles , so that your code looks like this:

|  |
| --- |
| public void SelectWeapon(int index)  {  currentWeaponIndex = index;  Player.SwitchWeapon(weapons[currentWeapon]);  }  public void Shoot()  {  Fire(bullets[currentWeapon]);  FireParticles(particles[currentWeapon]);  } |

The problem for this is not so much in the code, but rather setting it up in the inspector without making mistakes.

Rather, define a class that encapsulates the three variables, and make an array of that:

|  |
| --- |
| [Serializable]  public class Weapon  {  public GameObject prefab;  public ParticleSystem particles;  public Bullet bullet;  } |

The code looks neater, but most importantly, it is harder to make mistakes in setting up the data in the inspector.

**Avoid using arrays for structure other than sequences.**For example, a player may have three types of attacks. Each uses the current weapon, but generates different bullets and different behaviour.

You may be tempted to dump the three bullets in an array, and then use this kind of logic:

|  |
| --- |
| public void FireAttack()  {  /// behaviour  Fire(bullets[0]);  }  public void IceAttack()  {  /// behaviour  Fire(bullets[1]);  }  public void WindAttack()  {  /// behaviour  Fire(bullets[2]);  } |

Enums can make things look better in code…

|  |
| --- |
| public void WindAttack()  {  /// behaviour  Fire(bullets[WeaponType.Wind]);  } |

…but not in the inspector.

It’s better to use separate variables so that the names help show which content to put in. Use a class to make it neat.

|  |
| --- |
| [Serializable]  public class Bullets  {  public Bullet FireBullet;  public Bullet IceBullet;  public Bullet WindBullet;  } |

This assumes there is no other Fire, Ice and Wind data.

**Group data in serializable classes to make things neater in the inspector.** Some entities may have dozens of tweakables. It can become a nightmare to find the right variable in the inspector. To make things easier, follow these steps:

* Define separate classes for groups of variables. Make them public and serializable.
* In the primary class, define public variables of each type defined as above.
* Do not initialize these variables in Awake or Start; since they are serializable, Unity will take care of that.
* You can specify defaults as before by assigning values in the definition;

This will group variables in collapsible units in the inspector, which is easier to manage.

|  |
| --- |
| [Serializable]  public class MovementProperties //Not a MonoBehaviour!  {  public float movementSpeed;  public float turnSpeed = 1; //default provided  }  public class HealthProperties //Not a MonoBehaviour!  {  public float maxHealth;  public float regenerationRate;  }  public class Player : MonoBehaviour  {  public MovementProperties movementProeprties;  public HealthPorperties healthProeprties;  } |

**Tag/Layer:** One of the first things you should do is to create two static classes with your Tags and Classes so you don’t mess things up with upper/lower case problematic or use stuff like that from memory.

**Tag/Layer:** When checking a tag use the method CompareTag on Component or GameObject.

**Text**

**If you have a lot of story text, put it in a file.** Don’t put it in fields for editing in the inspector. Make it easy to change without having to open the Unity editor, and especially without having to save the scene.

**If you plan to localise, separate all your strings to one location.**There are many ways to do this. One way is to define a Text class with a public string field for each string, with defaults set to English, for example. Other languages subclass this and re-initialize the fields with the language equivalents.

More sophisticated techniques (appropriate when the body of text is large and / or the number of languages is high) will read in a spread sheet and provide logic for selecting the right string based on the chosen language.

**Testing and Debugging**

**Implement your own FPS counter.** Yup. No one knows what Unity’s FPS counter really measures, but it is not frame rate. Implement your own so that the number can correspond with intuition and visual inspection.

**Implement shortcuts for printing the player’s world position.** This makes it easy to report the position of bugs that occur in specific places in the world, which in turns makes it easier to debug.

**Implement debug options for making testing easier.** Some examples:

* Unlock all items.
* Disable enemies.
* Disable GUI.
* Make player invincible.
* Disable all gameplay.

**For teams that are small enough, make a prefab for each team member with debug options.** Put a user identifier in a file that is not committed, and is read when the game is run. This why:

* Team members do not commit their debug options by accident and affect everyone.
* Changing debug options don’t change the scene.

**Maintain a scene with all gameplay elements.**For instance, a scene with all enemies, all objects you can interact with, etc. This makes it easy to test functionality without having to play too long.

**Define constants for debug shortcut keys, and keep them in one place.**Debug keys are not normally (or conveniently) processed in a single location like the rest of the game input. To avoid shortcut key collisions, define constants in a central place. An alternative is to process all keys in one place regardless of whether it is a debug function or not. (The downside is that this class may need extra references to objects just for this).

**Documentation**

**Document your setup.** Most documentation should be in the code, but certain things should be documented outside code. Making designers sift through code for setup is time-wasting. Documented setups improved efficiency (if the documents are current).

Document the following:

* Layer uses (for collision, culling, and raycasting – essentially, what should be in what layer).
* Tag uses.
* GUI depths for layers (what should display over what).
* Scene setup.
* Idiom preferences.
* Prefab structure.
* Animation layers.

**Naming Standard and Folder Structure**

**Follow a documented naming convention and folder structure.**Consistent naming and folder structure makes it easier to find things, and to figure out what things are.

**Naming General Principles**

1. Call a thing what it is. A bird should be called Bird.
2. Choose names that can be pronounced and remembered. If you make a Mayan game, do not name your level QuetzalcoatisReturn.
3. Be consistent. When you choose a name, stick to it.
4. Use Pascal case, like this: ComplicatedVerySpecificObject. Do not use spaces, underscores, or hyphens, with one exception (see *Naming Different Aspects of the Same Thing*).
5. Do not use version numbers, or words to indicate their progress (WIP, final).
6. Do not use abbreviations: DVamp@W should be DarkVampire@Walk.
7. Use the terminology in the design document: if the document calls the die animation*Die*, then use DarkVampire@Die, not DarkVampire@Death.
8. Keep the most specific descriptor on the left: DarkVampire, not VampireDark;PauseButton, not ButtonPaused. It is, for instance, easier to find the pause button in the inspector if not all buttons start with the word *Button*. [Many people prefer it the other way around, because that makes grouping more obvious visually. Names are not for grouping though, folders are. Names are to distinguish objects of the same type so that they can be located reliably and fast.]
9. Some names form a sequence. Use numbers in these names, for example, PathNode0,PathNode1. Always start with 0, not 1.
10. Do not use numbers for things that don’t form a sequence. For example, Bird0, Bird1,Bird2 should be Flamingo, Eagle, Swallow.
11. Prefix temporary objects with a double underscore \_\_Player\_Backup.

**Naming Different Aspects of the Same Thing**

Use underscores between the core name, and the thing that describes the “aspect”. For instance:

* **GUI buttons states** EnterButton\_Active, EnterButton\_Inactive
* **Textures** DarkVampire\_Diffuse, DarkVampire\_Normalmap
* **Skybox** JungleSky\_Top, JungleSky\_North
* **LOD Groups** DarkVampire\_LOD0, DarkVampire\_LOD1

Do not use this convention just to distinguish between different types of items, for instance Rock\_Small, Rock\_Large should be SmallRock, LargeRock.

**Structure**

The organisation of your scenes, project folder, and script folder should follow a similar pattern.

**Folder Structure**

Materials

GUI

Effects

Meshes

Actors

DarkVampire

LightVampire

...

Structures

Buildings

...

Props

Plants

...

...

Plugins

Prefabs

Actors

Items

...

Resources

Actors

Items

...

Scenes

GUI

Levels

TestScenes

Scripts

Textures

GUI

Effects

...

**Scene Structure**

Cameras

Dynamic Objects

Gameplay

Actors

Items

...

GUI

HUD

PauseMenu

...

Management

Lights

World

Ground

Props

Structure

...

**Scripts Folder Structure**

ThirdParty

...

MyGenericScripts

Debug

Extensions

Framework

Graphics

IO

Math

...

MyGameScripts

Debug

Gameplay

Actors

Items

...

Framework

Graphics

GUI

...

**Optimizations**

**Cache component lookups**

A simple way to optimize component lookups that occur often is to cache the result. This is helpful for components that don’t change ever but are used often.

For example, when you do this:

using UnityEngine;

using System.Collections;

public class example : [MonoBehaviour](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html) {

void Update() {

transform.Translate(0, 0, 5);

}

}

Add this:

using UnityEngine;

using System.Collections;

public class example : [MonoBehaviour](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html) {

Transform thisTransform;

public new Transform transform {

get {

if (thisTransform == null) {

thisTransform = base.transform;

return thisTransform;

}

}

void Update() {

transform.Translate(0, 0, 5);

}

}

The latter code will run a lot faster since Unity doesn’t have to find the transform component in the game object each frame. The same applies for scripted components, where you use GetComponent instead of the transform or other shorthand property.

**Use Builtin arrays**

Builtin arrays are **fast**, very fast, so use them.

While the ArrayList or [Array](http://docs.unity3d.com/Documentation/ScriptReference/Array.html) classes are easier to use since you can easily add elements they don’t have nearly the same speed. Builtin arrays have a fixed size but most of the time you know the maximum size in advance and can just fill it out later. The best thing about builtin arrays is that they directly embed struct data types in one tightly packed buffer, without any extra type information or other overhead. Thus iterating through is very easy on the cache as everything is linear in memory.

Generic List (List<Object>) is apparently also very quick and should be used in every case where you need something dynamic like an ArrayList.

using UnityEngine;

using System.Collections;

public class example : [MonoBehaviour](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html) {

private [Vector3](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html)[] positions;

void Awake() {

positions = new [Vector3](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html)[100];

int i = 0;

while (i < 100) {

positions[i] = [Vector3.zero](http://docs.unity3d.com/Documentation/ScriptReference/Vector3-zero.html?from=index);

i++;

}

}

}

**Don’t call a function if you don’t have to**

The simplest and best of all optimizations is to perform less work. For example , when an enemy is far away it is most of the time perfectly acceptable to have the enemy fall asleep. That is do nothing until the player comes close.

Use triggers. A simple sphere trigger can work wonders though. You get OnTriggerEnter/Exit calls when exiting the sphere of influence you want

using UnityEngine;

using System.Collections;

public class example : [MonoBehaviour](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html) {

void OnTriggerEnter([Collider](http://docs.unity3d.com/Documentation/ScriptReference/Collider.html) c) {

if (c.CompareTag("Player"))

enabled = true;

}

void OnTriggerExit([Collider](http://docs.unity3d.com/Documentation/ScriptReference/Collider.html) c) {

if (c.CompareTag("Player"))

enabled = false;

}

}

**Don’t use Enum.ToString() in order to go from an Enum to a string.** Enums are very unreliable in terms of speed when doing this. You should instead use Enum.GetName(typeof(AttackType), AttackType.Melee); Tests shows that Enum.GetName is twice as fast as ToString.

**When casting a variable use the post fix “as type” instead of pre fixing with (type) as this is faster.** You loose the automatic null checking though. Remember that you don’t need to cast if you use the generic GetComponent method.

**Don’t use GameObject.Find to locate stuff, have it as a SerializeField and drag it in there instead if you can.** Your designers are going to love you for this in the end, even if is a bitch to set up at times.

**Let the OnBecameVisible and OnBecameInvisible methods in a MonoBehaviour be your friend.**

**Don’t throw Exceptions, they are slow.**

**The Transform class inherits from IEnumerable, which means you can easily loop through the children in a foreach.** Just remember that foreach sentences declares a variable that adds to your memory and that memory is not released on iOS, so it creates a memory leak.

**If you ever create dynamic textures or meshes, make sure to mark them as “HideFlags.DontSave”, or you will get the “Cleaning up leaked objects” message every time you save.**

**If you rely on SendMessage, be sure to check for whether the script is enabled inside your function, as even disabled scripts get messages.**

**Debugging**

**When debugging, the Debug.Break() is extremely useful for pausing the game at key frames.**

**Reducing Memory Usage**

***Avoid using foreach()***. It calls GetEnumerator() on your list type, which will allocate an enumerator on the heap just to throw it away. You’ll have to use the more verbose C++-style for(;;) syntax.

***Avoid strings***. Strings are immutable in .NET and allocated on the heap. You can’t manipulate them in-place like C. For UI, use [StringBuilders](http://msdn.microsoft.com/en-us/library/2839d5h5(v=vs.71).aspx) to build up strings in a memory-efficient manner, delaying the conversion to string until as late as possible. You can use them as keys because literals should point to the same instance in memory, but don’t manipulate them too much.

***Use structs***. Struct types in mono are allocated on the stack, so if you have a utility class that won’t leave scope, make it a struct. Remember structs are passed by value, so you will have to prefix a parameter with ref to avoid any copy costs.

***Replace scope-bound fixed-size arrays with structs***. If you have a fixed-size array that doesn’t leave scope, consider either replacing it with a member array that you can reuse, or create a struct with fields that mirror it. I replaced a Vector3[4] that was allocated every time you called our spline class with a ControlList struct that had four fields. I then added an this[] property for index-based access. This saved a ton of allocations because it was such a high-frequency function.

***Avoid IEnumerable extension methods***. It goes without saying that most Linq IEnumerable extension methods, as handy as they are, will create new allocations. However I was surprised that .Any(), called on an IList<>, which I expected to just be a virtual function to Count > 0, triggered an allocation. It’s the same for other funcs that should be trivial on an IList, like First() and Last(). If anyone can illuminate me on why this is I’d appreciate it. Because of this, and the foreach() restriction, I’d go as far as to say avoid the IEnumerable<> abstraction in your interfaces, and use IList<> instead.

***Minimise use of function* pointers.** Putting a class method in a delegate or a Func<> causes it to be boxed, which triggers an allocation. I can’t find any way to store a link to a method without boxing. I’ve left most of my function pointers in there because they’re a massive boon to decoupling, and I’ll just have to live with the allocs, but I’ve removed some.

***Beware cloned* materials.** If you get the material property of any renderer, the material will be cloned even if you don’t set anything on it. This material isn’t GC’d, and is only cleared up when you either change levels or call Resources.UnloadUnusedAssets(). Use myRenderer.sharedMaterial if you know you don’t want to adjust the material.

**Android SDK Setup**

Never install the Java Development Kit, I don’t know what they do with it, but by putting java.exe in at least three places it keeps the Android SDK from ever finding a proper version, even by setting JAVA\_HOME or the Path variable. Don’t install the Android ADT package, look for the “Use and existing IDE” link and then download the SDK to install.

1. Install the Java Runtime Environment (JRE)
2. Install the Android SDK, at the end let it run the SDK Manager
3. In the SDK Manager install the defaults already checked (optionally you can turn off things like documentation and samples, but make sure you install the Platform Tools and USB Drivers). Make sure you at least add the version of Android installed on your device.
4. Turn on “USB Debugging” on your device. On a lot of phones this option is hidden, follow these steps to turn this option on.
   1. In the settings menu, find the “About phone” options, tap to go into that screen
   2. Find the “Software Information” option and tap it
   3. Find the “Build number” line and tap that about 5 times, you should start to see a countdown something like “3 more taps to become a developer” which counts down with each press until it says “You are now a developer”
   4. There will now be a new “Developer Options” menu in the settings (where About phone was). Tap it
   5. There should be an option for “USB Debugging”, enable it. If it’s greyed out, it might that you currently have the device connected via USB, disconnect it, enable the option and reconnect.
5. Open the console and run “adb start-server”
6. Connect your device
7. In the Console run “adb devices”, if your device is listed without the word “unauthorized” you are good to move on, otherwise:
   1. If your device is not listed from “adb devices” you may need to download special drivers for your device (I did for the LG G3). Just search the web for them, they should be easy to find I.E. “lg g3 adb drivers”.
   2. If your device is listed but says “unauthorized” you can try reconnecting, but it might be the USB port or your cable. I strongly suggest using the USB cable that comes with your device. Sadly not all USB cables are created equal and I had to use the one that came with my phone before it recognized properly. When I used the correct cable a prompt came up on the phone to authorize the computer, accept it and move on. You might want to try the cable on different ports if this doesn’t happen.
8. You will probably have to restart Unity3D if it was open when you first connected the device.
9. In Unity3D select Edit->Preferences, on the External Tools tab click the Android SDK Location and select the folder that has the “SDK Manager.exe” file in it, along with other folders, etc.
10. From the file->build settings window click Android and click “Switch Platform”
11. You should be able to do “Build and run” at this point”

**Windows Phone 8 links**

I haven’t had time to go into these pages and find the relevant tips, but I also don’t want to lose these links in an email somewhere.

<http://blogs.msdn.com/b/uk_faculty_connection/archive/2013/09/11/getting-started-building-windows-phone-games-with-unity.aspx>

<http://www.c-sharpcorner.com/UploadFile/99bb20/extended-splash-screen-in-metro-style-application/>

<http://sertacozercan.com/2013/12/how-to-add-extended-splash-screen-with-progress-rings-to-a-windows-8-app/>

<http://developer.nokia.com/community/wiki/Custom_splash_screen_with_progress_bar_for_Windows_Phone_applications>

<http://developer.nokia.com/community/wiki/Splash_Screen_with_additional_splash_image_on_Windows_Phone>

<http://darkgenesis.zenithmoon.com/further-adventures-in-unhandled-exception-handling-for-win8/>

**Interesting gotchas I don’t know where else to put**

**From the Unity Blog:**

**Custom classes behave like structs**

|  |
| --- |
| [Serializable]  class Animal  {  public string name;  }  class MyScript : MonoBehaviour  {  public Animal[] animals;  } |

If you populate the animals array with three references to a single Animal object, in the serializationstream you will find 3 objects. When it’s deserialized, there are now three different objects. If you need to serialize a complex object graph with references, you cannot rely on Unity’s serializer doing that all automagically for you, and have to do some work to get that object graph serialized yourself. See the example below on how to serialize things Unity doesn’t serialize by itself.

Note that this is only true for custom classes, as they are serialized “inline” because their data becomes part of the complete serializationdata for the MonoBehaviour they are used in. When you have fields that have a reference to something that is a UnityEngine.Object derived class, like a “public Camera myCamera”, the data from that camera are not serialized inline, and an actual reference to the camera UnityEngine.Object is serialized.

**No support for null for custom classes**

Pop quiz. How many allocations are made when deserializing a MonoBehaviour that uses this script:

|  |
| --- |
| class Test : MonoBehaviour  {  public Trouble t;  }  [Serializable]  class Trouble  {  public Trouble t1;  public Trouble t2;  public Trouble t3;  } |

It wouldn’t be strange to expect 1 allocation, that of the Test object. It also wouldn’t be strange to expect 2 allocations, one for the Test object and one for a Trouble object. The correct answer is **729**. The serializer does not support null. If it serializes an object and a field is null, we just instantiate a new object of that type and serialize that. Obviously this could lead to infinite cycles, so we have a relatively magical depth limit of 7 levels. At that point we just stop serializing fields that have types of custom classes/structs and lists and arrays. [1]

Since so many of our subsystems build on top of the serialization system, this unexpectedly big serializationstream for the Test monobehaviour will cause all these subsystems to perform more slowly than necessary. When we investigate performance problems in customer projects, we almost always find this problem and we added a warning for this situation in Unity 4.5.

[1] I lied, the correct answer isn’t actually 729. This is because in the very very old days before we had this 7 level depth limit, Unity would just endless loop, and then run out of memory if you created a script like the Trouble one I just wrote. Our very first fix for that 5 years ago was to just not serialize fieldtypes that were of the same type as the class itself. Obviously, this was not the most robust fix, as it’s easy to create a cycle using Trouble1->Trouble2->Trouble1->Trouble2 class. So shortly afterwards we actually implemented the 7 level depth limit to catch those cases too. For the point I’m trying to make however it doesn’t matter, what matters is that you realize that if there is a cycle you are in trouble.

**No support for polymorphism**

If you have a

|  |
| --- |
| public Animal[] animals |

and you put in an instance of a dog, a cat and a giraffe, after serialization, you will have three instances of Animal.

One way to deal with this limitation is to realize that it only applies to “custom classes”, which get serialized inline. References to other UnityEngine.Object’s get serialized as actual references and for those, polymorphism does actually work. You’d make a ScriptableObject derived class or another MonoBehaviour derived class, and reference that. The downside of doing this, is that you need to store that monobehaviour or scriptable object somewhere and cannot serialize it inline nicely.

Sources:

<http://devmag.org.za/2012/07/12/50-tips-for-working-with-unity-best-practices/>

<https://docs.google.com/document/d/1qrAcSyLoiVBNzOiJYlC64UiU3ZwDbsw-zSzIeRzZnnE/edit>

<http://docs.unity3d.com/Documentation/ScriptReference/index.Performance_Optimization.html>

<http://andrewfray.wordpress.com/2013/02/04/reducing-memory-usage-in-unity-c-and-netmono/>

<http://blogs.unity3d.com/2014/06/24/serialization-in-unity/>