GameDev.tv Best Practices for Unity

Derived from blog by Herman Tulleken

Herman published the original *50 Tips for working with Unity* about 4 years ago.

Although a lot of it is still relevant, a lot has changed since then:

- Unity got better. For example, I now trust the FPS counter. The ability to use property drawers makes it less necessary to write custom editors. The way prefabs work makes the need for explicit nested prefabs or an alternative less. Scriptable objects are friendlier.
- Visual Studio integration got better, making it a lot easier to debug, and reducing the need for so much gorilla debugging.
- Third-party tools and libraries got better. There are now many assets in the Asset Store available that takes care of things such as visual debugging and better logging. Our own (free) *Extensions* plug-in has much of the code described in the original article (and a lot of it described here too).
- Version control got better. (Or perhaps, now I know better how to use it effectively). No need to have multiple copies or backup copies of prefabs, for example.

- I got more experience. In the last 4 years I worked on many Unity projects; including tons of game prototypes, production games such as Father.IO, and our flagship Unity asset *Grids*.

This article is a revised version of the original taking all of the above into account.

Before getting on with the tips, here is a disclaimer (essentially the same as the original).

These tips don't apply to every Unity project:

- They are based on my experience with projects with small teams from 3 to 20 people.
- There's is a price for structure, re-usability, clarity, and so on — team size, project size, and project goal determine whether that price should be paid. You would not use all these during a game jam, for example.
- Many tips are a matter of taste (there may be rivalling but equally good techniques for any tip listed here).

Unity also has some best practices on their site (although these are mostly from a performance point of view):

- http://unity3d.com/learn/tutorials/topics/best-practices
- Best practices for physically based content creation https://youtu.be/OeFYFUCa4tI
- 2D Best practices in Unity https://youtu.be/HM17mAmLd7k
- Internal Unity tips and tricks https://youtu.be/Ozc_hXzp_KU
- Unity Tips and Tricks https://youtu.be/2S6Ygq58QF8
- http://docs.unity3d.com/Manual/HOWTO-ArtAssetBestPracticeGuide.html

Workflow

1. Decide on the scale from the beginning and build everything to the same scale. If you don't, you may have need to rework assets later (for example, animation does not always scale correctly). For 3D games, using 1 Unity unit = 1m is usually the best. For 2D games that does not use lighting or physics, 1 Unity unit = 1 pixel (at "design" resolution) is usually good. For UI (and 2D games), pick a design resolution (we use HD or 2xHD) and design all assets to scale in that resolution.

2. Make every scene runnable. Do this to avoid having to switch scenes to run the game so that you can test faster. This can be tricky if you have objects that persist between scene loads that is required in all your scenes. One way of doing this is to use make persistent objects singletons that will load themselves when they are not present in the scene. Singletons are described in more detail in another tip.

3. Use source control and learn how to use it effectively.

- Serialize your assets as text. It does not in practice really make scenes and prefabs more mergeable, but it does make it easier to see what changed.
- Adopt a scene and prefab sharing strategy. In general, more than one person should not work on the same scene or prefab. For a small team it may be enough to ask around that no-one else is working on a scene or prefab before starting to work on it. It may be useful to swap physical tokens that denote scene ownership around (you are only allowed to work on a scene if you have the scene token on your desk).
- Use tags as bookmarks.

- Decide and stick to a branching strategy. Because scenes and prefabs cannot be smoothly merged, branching is slightly more complicated. However you decide to use branches, it should work with your scene and prefab sharing strategy.
- Use submodules with care. Submodules can be a great way to maintain re-usable code. But there are a few caveats:
    - Meta-files are not generally consistent over multiple projects. This is not generally a problem for non-Monobehaviour or non-Scriptable object code, but for MonoBehaviours and Scriptable objects using submodules can cause code to get lost.
    - If you work on many projects (including one or more in submodules), you can sometimes get an update avalanche where you have to have to pull-merge-commit-push over various projects for a few iterations to stabilize the code over all projects (and if someone else is making changes while this is going on, it could turn into a sustained avalanche). One way to minimize this effect is to always make changes to submodules from projects dedicated to them. This way, projects that use submodules only always have to pull; they never need to push back.

4. Keep test scenes and code separate. Commit temporary assets and scripts to the repository, and remove them from the project when you are done.

5. If you upgrade tools (especially Unity), do so simultaneously. Unity is much better at preserving links when you open a project with a different version than it used to be, but links still sometimes get lost when people are working with different versions.

6. Import third-party assets in a clean project and export a new package for your own use from there. Assets can sometimes cause problems when you import them directly in you project:

- There may be collisions (file or name), especially assets that have files in the root of the Plugins folder, or that use assets from Standard Assets in their examples.
- They may be unorganized on put their files all over your own project. This is especially a problem if you decide not to use it and want to remove it.

Follow these steps to make importing assets safer:

1. Make a new project, and import the asset.
2. Run the examples and make sure they work.
3. Organize the asset into a more suitable folder structure. (I usually do not enforce my own folder structure on an asset. But I make sure that all the files are in a single folder, and that there are not any files in important places that could overwrite existing files in my project.)
4. Run the examples and make sure they still work. (On occasion, I had assets break when I move stuff, but generally this should not be a problem).
5. Now remove all the things you won't need (such as examples).
6. Make sure the asset still compiles and that prefabs still have all their links. If there is anything left to run, test it.
7. Now select all the assets, and export a package.
8. Import into your project.

7. Automate your build process. This is useful even for small projects, but it is particularly useful when:

- you need to build lots of different versions of the game,

- other team-members with varying degrees of technical knowledge need to make builds, or
- you need to make small tweaks to the project before you can build.

See Unity Builds Scripting: Basic and advanced possibilities for a good guide on how to do this.

8. 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:

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

General Coding

9. Put all your code in a namespace. This avoids code clashes among your own libraries and third-party code. But don't rely on namespaces to avoid clashes with important classes. Even if you use different namespaces, don't use "Object" or "Action" or "Event" as class names.

10. Use assertions. Assertions are useful to test invariants in code and help flush out logic bugs. Assertions are available in the Unity.Assertions.Assert class. They all test some condition, and write an error message in the console if the condition is not met. If you are not familiar with how assertions can be useful, see *The Benefits of programming with assertions (a.k.a. assert statements)*.

11. Don't use strings for anything other than displayed text. In particular, do not use strings for identifying objects or prefabs. There are exceptions (there are still a few things that can only be accessed by name in Unity). In such cases, define those strings as constants in files such as AnimationNames or AudioModuleNames. If these classes become unmanageable, use nested classes so you can say something like AnimationNames.Player.Run.

12. Don't use Invoke and SendMessage. These methods of MonoBehaviour call other methods by name. Methods called by name is hard to track in code (you cannot find "Usages", and SendMessage has a wide scope that is even harder to track).

It is easy to roll out your own Invoke using Coroutines and C# actions:

```
public static Coroutine Invoke(this MonoBehaviour monoBehaviour, Action action, float time)
{
  return monoBehaviour.StartCoroutine(InvokeImpl(action, time));
}

private static IEnumerator InvokeImpl(Action action, float time)
```

```
{
  yield return new WaitForSeconds(time);

  action();
}
```

You can use this then like this in your monoBehaviour:

this.Invoke(ShootEnemy); //where ShootEnemy is a parameterless void method.

If you implement your own base MonoBehaviour, you can add your own Invoke to that.

A safer SendMessage alternative is more difficult to implement. Instead, I usually use GetComponent variaties to get components on parents, the current game object, or children, and make the call directly.

(Edit: Someone suggested ExecuteEvent, part of Unity's Event System, as an alternative. I could not find out much about it so far, but it looks worth investigating further.)

13. 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 an empty game object, or even a singleton (see later in this article) with no behaviour to make it easier to access from code. Call this object DynamicObjects.

14. Be specific about using null as a legal value, and avoid it where you can.

Nulls are helpful in detecting incorrect code. However, if you make if habit of silently passing over null, your incorrect code will happily run and you won't notice the bug until much later. Moreover, it can manifest itself deep in the code as each layer passes over null variables. I try to avoid using null as a legal value altogether.

My preferred idiom is to not do any null checking, and let the code fail where it is a problem. In methods that serve as an interface to a deeper level, I will check a variable for null and throw an exception instead of passing it on to other methods where it may fail.

In some cases, a value can be legitimately null, and needs to be handled in a different way. In cases like this, add a comment to explain when and why something can be null.

A common scenario is often used for inspector-configured values. The user *can* specify a value, but if she doesn't, a default value is used. A better way to do this is with a class Optional<T> that wraps values of T. (It's a bit like Nullable<T>). You can use a special property renderer to render a tick box and only show the value box if it is ticked. (Unfortunately, you cannot use the generic class directly, you have to extend classes for specific values of T.)

```
[System.Serializable]
public class Optional<T>
{
  public bool useCustomValue;
```

```
  public T value;
}
```

In your code, you can then use it like this:

```
health = healthMax.useCustomValue ? healthMax.Value : DefaultHealthMax;
```

Edit: Many people pointed out that it is better to use a struct (does not generate garbage, and cannot be null). However, this means you cannot use it as baseclass for non-generic classes so that you can use it for fields that can actually be used in the inspector.

15. If you use Coroutines, learn to use them effectively. Coroutines can be a powerful way to solve many problems. But they are hard to debug, and you can easily code yourself into a mess that no-one, not even yourself, can understand.

You should know:

- How to execute coroutines in parallel.
- How to execute coroutines in sequence.
- How to make new coroutines from existing ones.
- How to make custom coroutines using CustomYieldInstruction.

```
//This is itself a coroutine
IEnumerator RunInSequence()
{
  yield return StartCoroutine(Coroutine1());
  yield return StartCoroutine(Coroutine2());
}

public void RunInParallel()
{
  StartCoroutine(Coroutine1());
  StartCoroutine(Coroutine1());
}

Coroutine WaitASecond()
{
  return new WaitForSeconds(1);
}
```

16. Use extensions methods to work with components that share an interface. (Edit: Apparently GetComponent and so on now also work for interfaces, making this tip redundant). It is sometimes

convenient to get components that implement a certain interface, or find objects with such components.

The implementations below uses typeof instead of the generic versions of these functions. The generic versions don't work with interfaces, but typeof does. The methods below wraps this neatly in generic methods.

```
public static TInterface GetInterfaceComponent<TInterface>(this Component thisComponent)
  where TInterface : class
{
  return thisComponent.GetComponent(typeof(TInterface)) as TInterface;
}
```

17. Use extension methods to make syntax more convenient. For example:

```
public static class TransformExtensions
{
  public static void SetX(this Transform transform, float x)
  {
    Vector3 newPosition =
      new Vector3(x, transform.position.y, transform.position.z);

    transform.position = newPosition;
  }
  ...
}
```

18. Use a defensive GetComponent alternative. Sometimes forcing component dependencies through RequiredComponent is not always be possible or desirable, especially when you call GetComponent on somebody else's class. And even when you *do* use the RequiredComponent it is useful to indicate in the code where you get the component that you expect it to be there, and that it is a setup error when it is not. To do this, use an extension method that prints an error message or throws a more helpful exception when it is not found, like this one:

```
public static T GetRequiredComponent(this GameObject obj) where T : MonoBehaviour
{
  T component = obj.GetComponent();

  if(component == null)
  {
    Debug.LogError("Expected to find component of type "
      + typeof(T) + " but found none", obj);
  }
```

```
  return component;
}
```

19. Avoid using different idioms to do the same thing. In many cases there are more than one idiomatic way to do things. In such cases, choose one to use throughout the project. Here is why:

- Some idioms don't work well together. Using one idiom forces design in one direction that is not suitable for another idiom.
- Using the same idiom throughout makes it easier for team members to understand what is going on. It makes structure and code easier to understand. It makes mistakes harder to make.

Examples of idiom groups:

- Coroutines vs. state machines.
- Nested prefabs vs. linked prefabs vs. god prefabs.
- Data separation strategies.
- Ways of using sprites for states in 2D games.
- Prefab structure.
- Spawning strategies.
- Ways to locate objects: by type vs. name vs. tag vs. layer vs. reference ("links").
- Ways to group objects: by type vs. name vs. tag vs. layer vs. arrays of references ("links").
- Ways to call methods on other components.
- Finding groups of objects versus self-registration.
- Controlling execution order (Using Unity's execution order setup versus yield logic versus Awake / Start and Update / Late Update reliance versus manual methods versus any-order architecture).
- Selecting objects / positions / targets with the mouse in-game: selection manager versus local self-management.
- Keeping data between scene changes: through PlayerPrefs, or objects that are not Destroyed when a new scene is loaded.
- Ways of combining (blending, adding and layering) animation.
- Input handling (central vs. local)

20. Maintain your own time class to make pausing easier. Wrap Time.DeltaTime and Time.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).

Edit: Unity supports unscaledTime and unscaledDeltaTime, which makes having your own Time class redundant in many situations. It can still be useful when scaling global time affects components that you did not write in unwanted ways.

21. Custom classes that require updating should not access global static time. Instead, they should take delta time as a parameter of their Update method. This makes these classes useable when you implement a pausing system as explained above, or when you want to speed up or slow down the custom class's behavior.

22. Use a common structure for making WWW calls. In games with a lot of server-communication, it is common to have dozens of WWW calls. Whether you use Unity's raw WWW class or a plugin, you can benefit from writing a thin layer on top that does the boiler plate for you.

I usually define a Call method (one for each Get and Post), a CallImpl coroutine, and a MakeHandler. Essentially, the Call method builds a super hander from a parser, on-success and on-failure handlers using the MakeHandler method. It also calls the CallImpl courutine, which builds a URL, make the call, wait until it's done, and then call the super handler.

Here is roughly how it looks:

```
public void Call<T>(string call, Func<string, T> parser, Action<T> onSuccess, Action<string>
onFailure)
{
        var handler = MakeHandler(parser, onSuccess, onFailure);
        StartCoroutine(CallImpl(call, handler));
}

public IEnumerator CallImpl<T>(string call, Action<T> handler)
{
        var www = new WWW(call);
        yield return www;
        handler(www);
}

public Action<WWW> MakeHandler<T>(Func<string, T> parser, Action<T> onSuccess,
Action<string> onFailure)
{
  return (WWW www) =>
  {
    if(NoError(www))
    {
      var parsedResult = parser(www.text);
      onSuccess(parsedResult);
    }
    else
    {
      onFailure("error text");
    }
  }
}
```

This has several benefits.

- It allows you to avoid having to write a lot of boilerplate code.
- It allows you to handle certain things (such as displaying a loading UI component or handling certain generic errors) in a central place.

23. If you have a lot of 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.

24. If you plan to localize, 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.

Class Design

25. Decide how to implement inspectable fields, and make it a standard. There are two ways: make the fields public, or make them private and mark them as [SerializeField]. The latter is "more correct" but less convenient (and certainly not the method popularized by Unity itself). Whichever way you choose, make it a standard so that developers in your team know how to interpret a public field.

- Inspectable fields are public. In this scenario, public means "the variable is safe to change by a designer during runtime. Avoid setting its value in code".
- Inspectable fields are private and marked Serializable. In this scenario, public means "it's safe to change this variable in code" (and hence you should not see too many, and there should not be any public fields in MonoBehaviours and ScriptableObjects).

26. For components, never make variables public that should not be tweaked in the inspector. Otherwise they *will* be tweaked by a designer, especially if it is not clear what it does. In some rare cases it is unavoidable (for example, when some editor script needs to get hold of it). In that case you can use the HideInInspector attribute to hide it in the inspector.

27. Use property drawers to make fields more user-friendly. Property drawers can be used to customize controls in the inspector. This allows you to make controls that better fit the nature of the data, and put certain safe-guards in place (such as limiting the range of the variables). Use the Header attribute to organise fields, and use the Tooltip attribute to provide extra documentation to designers.

28. Prefer property drawers over custom editors. Property drawers are implement per field type, and is therefore much less work to implement. They are also more re-suable – once implemented for a type, they can be used for that type in any class. Custom editors are implemented per MonoBehaviour, and are therefor less re-usable and more work.

29. Seal MonoBehaviours by default. Generally, Unity's MonoBehaviours are not very inheritance-friendly:

- The way Unity calls message-methods like Start and Update makes it tricky to work with these methods in subclasses. If you are not careful, the wrong thing gets called, or you forget to call a base method.
- When you use custom editors, you usually need to duplicate the inheritance hierarchy for the editors. Anyone who wants to extend one of your classes has to provide their own editor, or make do with whatever you provided.

In cases where inheritance *is* called for, do not provide any Unity message-methods if you can avoid it. If you *do*, don't make them virtual. If necessary, you can define an empty virtual function that gets called from the message-method method that a child class can override to perform additional work.

```
public class MyBaseClass
{
  public sealed void Update()
  {
    CustomUpdate();
    ... // This class's update
  }

  //Called before this class does its own update
  //Override to hook in your own update code.
  virtual public void CustomUpdate(){};
}

public class Child : MyBaseClass
{
  override public void CustomUpdate()
  {
    //Do custom stuff
  }
}
```

This prevents a class from accidentally overriding your code, but still gives it the ability to hook into Unity's messages. One reason that I don't like this pattern is that the order of things becomes problematic. In the example above the child may want to do things directly after the class has done its own update.

30. Separate interface from game logic. Interface components should in general not know anything about the game in which they are used. Give them the data they need to visualize, and subscribe to events to find out when the user interacts with them. Interface components should *not* do gamelogic. They can filter input to make sure it's valid, but the main rule processing should happen elsewhere. In many puzzle-games, the pieces are an extension of the interface, and should not contain rules. (For example, a chess piece should not calculate its own legal moves.

Similarly, input should be separated from the logic that acts on that input. Use an input controller that informs your actor of the intent to move; the actor handles whether to actually move.

Here is a stripped down example of a UI component that allows the user to select a weapon from a list of choices. The only thing these classes know about the game is the Weapon class (and only because Weapon is a useful source for the data this container needs to display). The game also knows nothing about the container; all it has to do is register for the OnWeaponSelect event.

```
public WeaponSelector : MonoBehaviour
{
  public event Action OnWeaponSelect {add; remove; }
  //the GameManager can register for this event

  public void OnInit(List  weapons)
  {
    foreach(var weapon in weapons)
    {

      var button = ... //Instantiates a child button and add it to the hierarchy

      buttonOnInit(weapon, () => OnSelect(weapon));
      // child button displays the option,
      // and sends a click-back to this component
    }
  }
  public void OnSelect(Weapon weapon)
 {
    if(OnWepaonSelect != null) OnWeponSelect(weapon);
  }
}

public class WeaponButton : MonoBehaviour
{
   private Action<> onClick;

   public void OnInit(Weapon weapon, Action onClick)
   {
      ... //set the sprite and text from weapon

      this.onClick = onClick;
   }

   public void OnClick() //Link this method in as the OnClick of the UI Button component
   {
     Assert.IsTrue(onClick != null);  //Should not happen

     onClick();
   }
```

}

31. Separate configuration, state and bookkeeping.

- Configuration variables are the variables tweaked in the inspector to define your object through its properties. For example, *maxHealth*.
- State variables is the variables that completely determines your object's current state, and are the variables you need to save if your game supports saving. For example, *currentHealth*.
- Bookkeeping variables are used for speed, convenience, or transitional states. They can always completely be determined from the state variables. For example, *previousHealth*.

By separating these types of variables, you make it easier to know what you can change, what you need to save, what you need to send / retrieve over the network, and allows you to enforce this to some extent. Here is a simple example with this setup.

```
public class Player
{
  [Serializable]
  public class PlayerConfigurationData
  {
    public float maxHealth;
  }

  [Serializable]
  public class PlayerStateData
  {
    public float health;
  }

  public PlayerConfigurationData configuration;
  private PlayerState stateData;

  //book keeping
  private float previousHealth;

  public float Health
  {
    public get { return stateData.health; }
    private set { stateData.health = value; }
  }
}
```

32. 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.

33. 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.

34. 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;
}
```

35. Make classes that are not MonoBehaviours Serializable even when they are not used for public fields. This allows you to view the class fields in the inspector when the Inspector is in Debug mode. This works for nested classes too (private or public).

36. Avoid making changes to inspector-tweakables in code. A variable that is tweakable in the inspector is a configuration variable, and should be treated as a run-time constant and not double as a state-variable. Following this practice makes it easier to write methods to reset a component's state to the initial state, and makes it clearer what the variable does.

```
public class Actor : MonoBehaviour
{
  public float initialHealth = 100;

  private float currentHealth;

  public void Start()
  {
    ResetState();
  }

  private void Respawn()
  {
```

```
    ResetState();
  }

  private void ResetState()
  {
    currentHealth = initialHealth;
  }
}
```

Patterns

Patterns are ways to solve frequently occurring problems in a standard way. Bob Nystrom's book *Game Programming Patterns* (readable free online) is a useful resource to see how patterns apply to problems that arise in game programming. Unity itself use a lot of these patterns: Instantiate is an example of the prototype pattern; MonoBehaviours follow a version of the template pattern, UI and animation use the observer pattern, and the new animation engine uses state machines.

These tips relate to using patterns with Unity specifically.

37. 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.

(Many programmers warn against classes vaguely named *XManager* because it points at a class either poorly named, or designed with too many unrelated tasks. In general, I agree with this. However, we have a small number of managers in every game, and they do the same in every game, so that these classes are in fact idioms.)

- 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 your GameManager (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 of GameManager.Instance.player.

As explained in other tips, singletons are also useful for creating default spawn points and objects that persist between scene loads that keep track of global data.

38. Use state machines to get different behavior in different states or to execute code on state transitions. A light-weight state machine has a number of states, and for each state allows you to specify actions to run when entering or existing the state, and an update action. This can make code cleaner and less error prone. A good sign that you could benefit from a state machine is if your Update method's code has an if- or switch-statement that changes what it does, or variables such as hasShownGameOverMessage.

```
public void Update()
{
  if(health <= 0)
  {
    if(!hasShownGameOverMessage)
    {
      ShowGameOverMessage();
      hasShownGameOverMessage = true; //Respawning resets this to false
    }
  }
  else
  {
    HandleInput();
  }
}
```

With more states, this type of code can become very messy; a state machine can make it a lot cleaner.

39. Use fields of type UnityEvent to set up the observer pattern in the inspector. The UnityEvent class allows you to link in methods that take up to four parameters in the inspector using the same UI interface as the events on Buttons. This is especially useful for dealing with input.

40. Use the observer pattern to detect when a field value changes. The problem of executing code only when a variable changes crops up frequently in games. We have baked a general solution of this in a generic class that allows you to register for events whenever the value changes. Here is an example with health. Here is how it is constructed:

```
/*ObservedValue*/ health = new ObservedValue(100);
health.OnValueChanged += () => { if(health.Value <= 0) Die(); };
```

You can now change it everywhere, without doing checking in each place where you check it, for example, like this:

```
if(hit) health.Value -= 10;
```

Whenever the health hits a point below 0, the Die method is called. For further discussion and an implementation, see this post.

41. Use the Actor pattern on prefabs. (This is not a "standard" pattern. The basic idea is from Kieran Lord in this presentation.)

An actor is the main component on the prefab; usually the component that provides the prefab's "identity", and the one higher-level code will most often interact with. The actor uses other components – helpers – on the same object (and sometimes on children) to do its work.

If you make a button object from the menu in unity, it creates a game object with a Sprite and Button component (and a child with a Text component). In this case, Button is the actor component. Similarly, the main camera typically has several components (GUI Layer, Flare Layer, Audio Listener) in addition to the Camera component attached. Camera is the actor.

An actor may require other components to work correctly. You can make your prefab more robust and useful by using the following attributes on your actor component:

- Use RequiredComponent to indicate all the components your actor needs on the same game object. (Your actor can then always safely call GetComponent, without the need to check whether the value returned was null.)
- Use DisallowMultipleComponent to prevent multiple instances of the same component to be attached. Your actor can then always call GetComponent without having to worry what the behavior shoud be when there is more than one component attached).
- Use SelectionBase if your actor object has children. This makes it easier to select in the scene view.

```
[RequiredComponent(typeof(HelperComponent))]
[DisallowMultipleComponent]
[SelectionBase]
public class Actor : MonoBehaviour
{
  ...//
}
```

42. Use generators for random and patterned data streams. (This is not a standard pattern, but one we found to be extremely useful.)

A generator is similar to a random generator: it is an object with a Next method that can be called to get a new item of a specific type. Generators can be manipulated during their construction to produce a large variety of patterns or different types of randomness. They are useful, because they keep the logic of generating a new item separate from where you need the item, so that your code is much cleaner.

Here are a few examples:

```
var generator = Generator
  .RamdomUniformInt(500)
  .Select(x => 2*x); //Generates random even numbers between 0 and 998

var generator = Generator
  .RandomUniformInt(1000)
  .Where(n => n % 2 == 0); //Same as above

var generator = Generator
   .Iterate(0, 0, (m, n) => m + n); //Fibonacci numbers

var generator = Generator
  .RandomUniformInt(2)
  .Select(n => 2*n - 1)
  .Aggregate((m, n) => m + n); //Random walk using steps of 1 or -1 one randomly

var generator = Generator
  .Iterate(0, Generator.RandomUniformInt(4), (m, n) => m + n - 1)
  .Where(n >= 0); //A random sequence that increases on average
```

We have use generators for spawning obstacles, changing background colors, procedural music, generating letters sequences likely to make words in word games, and much more. Generators also work well to control co-routines that repeat at non-constant intervals, using the construct:

```
while (true)
{
  //Do stuff

  yield return new WaitForSeconds(timeIntervalGenerator.Next());
}
```

To find out more about generators, see this post.

Prefabs and Scriptable Objects

43. Use prefabs for everything. The only game objects in your scene that should not be prefabs (or part of 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.

44. 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.

45. 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 losing any values that you set up in the inspector.

46. Use scriptable objects for shared configuration data instead of prefabs.

If you do this:

- scenes are smaller
- you cannot make changes to a single scene (on a prefab instance) by mistake.

47. Use scriptable objects for level data. Level data is often stored in XML or JSON, but using scriptable objects instead has a few advantages:

- It can be edited in the Editor. This makes it easier to validate data and is friendlier to non-technical designers. Moreover, you can use custom editors to make it even easier to edit.
- You don't have to worry about reading / writing and parsing of the data.
- It is easier to split and nest, and manage the resulting assets, and so compose levels from building blocks rather than from a massive configuration.

48. Use scriptable objects to configure behavior in the inspector. Scriptable objects are usually associated with configuring data, but they also allow you to use "methods" as data.

Consider a scenario where you have an Enemy type, and each enemy has a bunch of SuperPowers. You could make these normal classes and have a list if them in the Enemy class… but without a custom editor you would not be able to set up a list of different superpowers (each with its own properties) in the inspector. But if you make these super powers assets (implement them as ScriptableObjects), you could!

Here is how it looks:

```
public class Enemy : MonoBehaviour
{
  public SuperPower superPowers;


  public UseRandomPower()
  {
     superPowers.RandomItem().UsePower(this);
  }
}

public class BasePower : ScriptableObject
{
  virtual void UsePower(Enemy self)
  {
  }
}

[CreateAssetMenu("BlowFire", "Blow Fire")
public class BlowFire : SuperPower
{
  public strength;
  override public void UsePower(Enemy self)
  {
    ///program blowing fire here
  }
}
```

There are a few things to be aware of when following this pattern:

- Scriptable objects cannot reliably be made abstract. Instead, use concrete bases classes, and throw NotImplementedExceptions in methods that *should* be abstract. You can also define an Abstract attribute and mark classes and methods that should be abstract with it.
- Scriptable objects that are generic cannot be serialized. However, you can use generic base classes and only serialize sub-classes that specify all the generics.

49. Use scriptable objects to specialize prefabs. If two objects' configuration differ only in some properties, it is common to put two instances in the scene and adjust those properties on the instances. It is usually better to make a separate class of the properties that can differ between the two types into a separate scriptable object class.

This gives you more flexibility:

- You can use inherit from your specialization class to give more specific properties to different types of object.
- Scene setup is much safer (you merely select the right scriptable object, instead of having to adjust all the properties to make the object into the desired type).
- It is easier to manipulate these objects at runtime through code.
- If you have multiple instances of the two types, you know their properties will always be consistent when you make changes.
- You can split sets of configuration variables into sets that can be mixed-and-matched.

Here is a simple example of this setup.

```
[CreateAssetMenu("HealthProperties.asset", "Health Properties")]
public class HealthProperties : ScriptableObject
{
  public float maxHealth;
  public float resotrationRate;
}

public class Actor : MonoBehaviour
{
  public HealthProperties healthProperties;
}
```

If the number of specializations is large, you may want to define the specialization as a normal class, and use a list of these in a scriptable object that's linked to a suitable place where you can get hold of it (such as your *GameManager*). There is a bit more glue necessary to make it safe, fast and convenient; just the bare minimum is shown below.

```
public enum ActorType
{
  Vampire, Wherewolf
}

[Serializable]
public class HealthProperties
{
  public ActorType type;
  public float maxHealth;
```

```
  public float resotrationRate;
}

[CreateAssetMenu("ActorSpecialization.asset", "Actor Specialization")]
public class ActorSpecialization : ScriptableObject
{
  public List healthProperties;

  public this[ActorType]
  {
     get { return healthProperties.First(p => p.type == type); } //Unsafe version!
  }
}

public class GameManager : Singleton
{
  public ActorSpecialization actorSpecialization;

  ...
}

public class Actor : MonoBehaviour
{
  public ActorType type;
  public float health;

  //Example usage
  public Regenerate()
  {
    health
      += GameManager.Instance.actorSpecialization[type].resotrationRate;
  }
}
```

50. Use the CreateAssetMenu attribute to add ScriptableObject creation automatically to the Asset/Create menu.

Debugging

51. Learn how to use Unity's debugging facilities effectively.

- Add context objects to Debug.Log statements to see from where they are generated.

- Use Debug.Break to pause the game in the editor (useful, for example, when you want to an error condition occurs and you want to examine component properties in that frame).
- Use Debug.DrawRay and Debug.DrawLine functions for visual debugging (for example, DrawRay is very useful when debugging why ray casts are not hit).
- Use Gizmos for visual debugging. You can also supply gizmo renderers *outside* mono behaviours using the DrawGizmo attribute.
- Use the *debug inspector view* (to see the values of private fields at runtime in Unity using the inspector).

52. Learn how to use your IDE debugger effectively. See for example Debugging Unity games in Visual Studio.

53. Use a visual debugger that draws graphs of values over time. This is extremely helpful to debug physics, animation, and other dynamic processes, especially sporadic glitches. You will be able to see the glitch in the graph, and be able to see which other variables change at the same time. The visual inspection also makes certain types of strange behavior clear, such as values that change too often, or drift without apparent cause. We use Monitor Components, but there are several available.

54. Use improved console logging. Use an editor extension that allows you to color-code output according to categories, and allows you to filter output according to those categories. We use Editor Console Pro, but there are several available.

55. Use Unity's test tools, especially to test algorithms and mathematical code. See for example the *Unity Test Tools* tutorial, or the post Unit testing at the speed of light with Unity Test Tools.

56. Use Unity's test tools to run "scratchpad" tests. Unity's test tools are not *only* suitable for formal tests. They can also be exploited for convenient scratch-pad tests that can be run in the editor without having to run a scene.

57. Implement shortcuts for taking screen shots. Many bugs are visual, and are much easier to report when you can take a picture. The ideal system should maintain a counter in PlayerPrefs so that successive screenshots are not overwritten. The screenshots should be saved outside the project folder to avoid people from accidentally committing them to the repository.

58. Implement shortcuts for printing snapshots of important variables. This makes it easy to log some information when something unexpected happens during the game that you can inspect. Which variables depends on the game, of course. You will be guided by the typical bugs that occur in your game. Examples are positions of the player and enemies, or the "thinking state" an AI actor (such as the path it is trying to follow).

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

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

Be careful not to commit debug options accidentally; changing debug options can mystify other developers on your team.

60. 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).

61. Draw or spawn small spheres at vertices when doing procedural mesh generation. This will help you make sure your vertices are where they are supposed to be and the mesh is the right size before you start messing with triangles and UVs to get your mesh to display.

Performance

62. Be wary of generic advice about design and construction for performance reasons.

- Such advice is often based on myths and is not backed by tests.
- Sometimes the advice is backed by tests but the tests are faulty.
- Sometimes the advice is backed by correct tests, but they are in an unrealistic or different context. (For example, it's easy to show how using arrays are faster than generic lists. However, in the context of a real game this difference is almost always negligible. Similarly, if the tests apply to different hardware than your target devices, their results may not be meaningful to you.)
- Sometimes the advice is sound, but out of date.
- Sometimes, the advice applies. However, there is a tradeoff. Slow games that ship are sometimes better than fast ones that don't. And heavily optimized games are more likely to contain tricky code that can delay shipping.

Performance advice *can* be useful to keep in mind to help you track the source of *actual* problems faster using the process outlined below.

63. Test regularly on target devices from early-on. Devices have very different performance characteristics; don't get surprised by them. The earlier you know about problems the more effectively you can address them.

64. Learn how to use a profiler effectively to track the cause of performance problems.

- If you are new to profiling, see Introduction to the Profiler.
- Learn how to define your own frames (using Profiler.BeginFrame and Profiler.EndFrame) for fain-grained analysis.
- Learn how to use platform-specific profiling, such as the built-in profiler for iOS.
- Learn to profile to file in built players and display the data in the profiler.

65. Use a custom profiler for more accurate profiling when necessary. Sometimes, Unity's profiler cannot give you a clear picture of what is going on: it may run out of profile frames, or deep-profiling can slow down the game so much that tests are not meaningful. We use our own in-house profiler for this, but you should be able to find alternatives on the Asset Store.

66. Measure impact of performance enhancements. When you make a change to increase performance, measure it to make sure the change is a real improvement. If the change is not measurable or negligent, undo it.

67. Don't write less-readable code for better performance. Unless:

- You have a problem, identified the source with a profiler, you measured a gain after the change, and the gain is high enough compared with the loss in maintainability.

OR

- You know what you are doing.

Naming Standard and Folder Structure

68. 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.

You will most probably want to create your own naming convention and folder structure. Here is one as an example.

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. Don't call something buttonHolder in one place and buttonContainer in another.
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 useDarkVampire@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 organization of your scenes, project folder, and script folder should follow a similar pattern. Here are some abridged examples to get you started.

Folder Structure

MyGame
  Helper
    Design
    Scratchpad
  Materials
  Meshes
    Actors
      DarkVampire
      LightVampire
      ...
    Structures
      Buildings
      ...
    Props
      Plants
      ...
    ...
  Resources
    Actors
    Items
    ...
  Prefabs
    Actors
    Items
    ...
  Scenes
    Menus
    Levels
  Scripts
  Tests
  Textures
    UI
    Effects
    ...
  UI
MyLibray

...
Plugins
SomeOtherAsset1
SomeOtherAsset2
...

Scene Structure

Main
Debug
Managers
Cameras
Lights
UI
  Canvas
    HUD
    PauseMenu
    ...
World
  Ground
  Props
  Structures
  ...
Gameplay
  Actors
  Items
  ...
Dynamic Objects

Scripts Folder Structure

Debug
Gameplay
  Actors
  Items
  ...
Framework
Graphics
UI
...