2014

C# Coding Standard

Lazy Dog Games

Table of Contents

Introduction

This coding standard has been written to ensure all game code written in C# adheres to the same basic guidelines. It is imperative that all code remains well organised and structured, so that it can be easily re-used in the future. By enforcing this basic standard, all future projects can be developed with ease as the code will be a lot more clearer to read, easier to debug and possible to extend.

There are many *do's* and *don'ts* in the programming field and it is important to be aware of these issues. While most of the practises in this document won't determine a program's fate, they will help you improve your programming style, and force you to remain consistent at all times. Better code leads to better software, so take the following ideas into consideration.

Good Coding Practise

No style of programming should be set in stone or obeyed as if it were a law, but rather as a general standard. There are some things that should be followed to make the code look neat, clear to read and understand, and avoid potential errors. Below are just a few simple tricks that can help make programming easier for you and save you many headaches.

- ***Use standard English*** : When naming your variables, methods, and classes, be sure to use standard English names, as this is the preferred language and it's international. This will ensure that a vast majority of programmers will be able to read and understand your code.

- ***Keep it tidy***: Although *Visual Studio* offers automatic indentation, some compilers may not, so make it a habit to indent all your code, as this will make your program more readable and clearer to others.
  When writing blocks of code, keep the open and closing braces lined up.
  When declaring and initializing variables, line up the data types, variables, assignment operators and values in a neat way.

- ***Comments*** : Commenting your code is important, as it makes things clear to other programmers what your intentions are or how a piece of code functions.
  However, don't comment the obvious and try to enforce

self-documenting code. This allows for minimal comments and maximum clarity.

- ***Code line limit*** : When coding in your IDE of choice, you may become accustomed to your widescreen monitor and write long lines of code across the screen, without realising that other programmers who will read and edit your code might not have a widescreen monitor at all. Therefore it is imperative to stick to a line limit, preferably no more than 100 lines across.

- ***Programming by intention*** : Sometimes you may want to write out all your classes and methods by only declaring them and creating empty code with comments, which you then fill in later on.
  When creating *generic* classes or methods, first create them by writing out the code as if it were a normal class or method, thereafter test it, and then replace all the data types with a ***<T>***.

- ***Compile & build often*** : While coding away, always be sure to compile and build your game or program as frequently as possible. This helps find errors early on in the development stage, rather that compiling a very large project that will reveal a long list of endless bugs.

- ***Don't ignore warnings*** : Even though some warnings may not always be serious, make sure you don't have any in your code. If you do, investigate and try to solve the issue rather than ignore it, as it could lead to bigger bugs later on.

- ***Use correct tools*** : Use the correct algorithm for the right calculation, or use the correct container for storing your data. Some containers prove expensive when used incorrectly and this can lead to poor game performance. When using loops, use the correct one for the task at hand.

- ***Things to avoid*** : There are a few things to steer clear from in order to avoid creating confusing and error prone code.
  Avoid using the *goto* statement. This can be handy to move to any place in your code, but it can become very confusing, break your program flow, and generally lead to errors.
  Try to avoid using *global variables*. Keep your variables as localized as possible, as this will also make things clear as to what the variable is intended to do. For more on variables, see the *Variables* section.
  Avoid *chaining assignments*, *nesting methods* too deeply and *implicit conversions* - instead make everything explicit by using a cast.
  Avoid using *infinite loops* like *for(; ;)* and *while(true).*
  Make use of parenthesis to explicitly state what you intend the code to do.

```
int x = (y * z) + (f * g);
```

- ***Braces*** : Use an open and closing brace, even if only one line of code is used, because it looks neater and allows for easily adding more code to the block later without re-typing too much.

- ***Debugging*** : When using *try-catch* blocks, always line up the most specific catch blocks before the generic ones. When catching certain errors, make sure you create meaningful error messages, that are clear to both fellow programmers and end-users.

- ***Ternary operator*** : Sometimes an *if-else* block of code is real simple and consists of one-liners. In this case make use of the Ternary operator, as it is easy to write and read if the statement is simple enough.

Variables

It is essential that variables are created, labelled and handled correctly to ensure sure less confusion and bugs when using them. They should never contain garbage or be used when out of scope. Below are a few pointers for using variables correctly.

- ***Variable naming*** : A good practise is to clearly label each variable with a descriptive name that is a maximum of 20 characters long. Sometimes names might contain multiple words, and in that case, it is advised to use *Camel Casing* in naming your variables. Sometimes variables are temporary, for instance if they are used in loops. In these cases, letters like ***x*** or ***i*** are sufficient.

  ```
  for(int i = 0; i < 10; i++)
  ```

  Give single-item variables singular names, like *score* or *playerName*. Give arrays or collection variables plural names, like *enemyNames* or *totalScores*.

- ***Immediate initialisation*** : Even though C# initialises variables to their default values, it is good practise to initialize them explicitly as soon as they are declared, or soon thereafter in an initializer method like *Start(), Awake()* or a class constructor. This makes it clear to other programmers what the variable is doing or going to do.

- ***Use correct data types*** : When deciding what type of variable you want to declare, consider what it will be used for. Try use only what is needed, so if you are storing a player health value, use unsigned variables, if you are working with time and gravity, use `float` or `double` . To preserve memory consider using `short` rather than `long` .
  When using floating point numbers use what's needed. The *decimal* type is highly accurate but heavy in calculations, so use it only when needed. Use *float* rather than *double*, unless high precision is required. By default all floating point numbers in C# are *double*, so append the ***f*** postfix for *float* variables.
  For the intrinsic data types, C# offers a corresponding class type. Therefore a string for instance could be declared as follows :

  ```
  string myString;
  String aString;
  ```

  When declaring variables for data types use *string*, *int*, *float*, etc instead of the classes *String*, *Int32*, *Double* etc

- ***Explicit declaration*** *:* When declaring variables, explicitly state their types. C# does provide a *var* and *dynamic* keyword for determining the type at *compile-time* and *runtime* respectively, however keep the code more readable by not using them.

- ***Localisation*** : Keep variables as local as possible. For instance, if a variable is being used in a class, consider whether to make it a field or a local variable in one of that class' methods. Avoid using global variables as much as you can. Usually, if you have enough data to initialize the variable, then declare it at that point.

- ***Constants*** : A good way to label constant variables is to name the variable entirely in capital letters. If multiple words and/or numbers are used, separate them with an underscore :

```
const float TOTAL_GRAVITY = 12.5f;
```

- ***Enumerations*** *:* Keep all enumerations inside a class declaration if they belong to a class, then they can be made *private* or *public* for outer access. Alternatively create them globally if they are going to be used globally. Declare enumerations in a single line, instead of a large block form, and use *Pascal Casing* to name the enumeration types and values :

```
public enum PlayerHealth  { Good, Bad, WorstYet };
```

Methods

It is important to break your code up into as many different sub-routines as possible. This is known as *refactoring*. Each individual task in the program or game should be handled by a method, and even some methods may want to call others to perform certain actions. If you find you have a lot of repetitive code in your program, consider using methods instead.

- ***Method naming*** : Use descriptive names for the methods, as this can make it clear as to what the method is designed to do. Start each name with a capital letter, and if the name consists of multiple words, use *Pascal Casing.* Separate words and numbers with an underscore.

- ***Use cohesion*** : Never give methods multiple tasks or too much of a task to handle. Instead make sure they perform one specific task. This is known as *cohesion*, and ensures that each method has a certain responsibility.

- ***Predicate methods*** **:** For predicate methods, name them all using **'is'** or **'has'** keywords, like *IsFullscreen().* This is essentially like a getter method, but it's clearer as to what it will return.

- ***Complimentary methods*** **:** Create methods that complement each other like *Initialize()* and *Shutdown()*

or *Load()* and *Unload().* What you do in the one you oppose in the other. This presents a good structure in the code and avoids confusion.

- ***Return statements*** : When using a void method, even though C# allows you to, don't use a *return* statement, just leave it out, because it adds unnecessary code.

Classes

Creating your own data types to break your game down into smaller bite-sized chunks is vital, and the key to OOP. C# and Unity focuses predominantly on classes, therefore the guidelines listed in this section are important.

- ***Class naming*** : All class names should begin with a capital letter and be labelled as *public* by default. If the name consists of multiple words, use *Pascal Casing* and separate words and numbers with an underscore.

- ***Encapsulation*** : Keep *encapsulation* in mind. Data hiding is important, so be sure to conceal as much of your class as possible. Start with *private* members, then work your way down the ladder of security to *protected*, *public*, etc. Good practise is to make all your data member variables *private* and use *properties* to gain indirect access to them from the client code.
  Keep all classes responsible for themselves. With regards to inheritance, each class should initialize and look after its own members, don't let sub-classes access base class members if it's not necessary.

- ***Good design*** : OOP is all about elegantly designing your classes to form a robust and well designed structure. Know when to use ***IS_A, HAS_A*** and ***CAN_BE_USED_AS***. Beware of *tight coupling* - sometimes classes that communicate with or use each other know too much about themselves. This design is weak and should be avoided. Consider using *factories* or *Interfaces* instead.

- ***Methods*** : Class methods should follow the same guidelines as global methods *(See Methods section).* When defining them, explicitly label methods with *public*, *private* or *protected*, except *Unity* specific methods.

- ***Fields*** : Class fields should follow the same guidelines as variables *(See Variables section).* When declaring them, list all *public* fields before the *private* ones, and explicitly label them *public*, *private* or *protected*. Only make fields *public* if they need to appear in the *Unity Inspector* or if they really need to have global access. Do not precede the field names with an ***m_*** prefix, as is done in C++.

- ***Properties*** : These special methods are a great way of accessing class members, and should be used externally as well as internally to access data members from inside the class itself. Be careful not to write properties if they are not needed.
  Use the shorthand method by default and only use the longer method if more complex code is needed. If the shorthand version is used, name the properties with a capital letter, and place the getter/setter statements all in one line.

  ```
  public int Number { get; set; }
  ```

  Note : The shorthand method will not work with the Unity inspector.

- ***Constructor guidelines*** : Make sure your constructor does all the data member initialization, and runs all the setup code, instead of using a separate *Init()* method for instance. Make sure fields are initialized in the same order as they appear in their declaration.
  If you are creating your own constructors and you create ones that take parameters, make sure you create default ones too, so the compiler doesn't complain if you ever create the object the default way.

- ***Destructor guidelines*** : Use your destructor to do any necessary cleanup.
  Note : With regards to C# and Unity, destructors aren't widely used

- ***Interfaces*** : Make sure these class types have an ***I*** preceding their name, and that the name of the interface class is an adjective, for instance *IMovable.*

- ***Unity classes*** : Just like you do with constructors in standard classes, make sure the *Start()* or *Awake()* methods do all the data member initialization and setup tasks.
  Create separate methods for specific functionality. Keep the *Start(), Update()* and other methods small – they should rather call these custom methods to do all the extra work.
  Place all *Unity* methods first, then follow those with the custom methods.

Resource files

Resource files are what the game engine feeds on to run properly. All resources like images, sounds, animations, models, etc are stored in these files and so it is important to create and use them properly.

- ***File naming*** : All asset files should be named using *Pascal Casing*, preceded by the category. If a number needs to be appended, use an underscore in between. For example :

  MainLevel-Background.png
  Hallway-EerieVoices_01.wav

  All script files should be named the exact same way as the classes that they contain. They should be named also using *Pascal Casing.* If a number needs to be appended, use an underscore in between. For example :

  MainPlayer.cs
  PlayerMove_01.cs

- ***Tidy project structure*** : Name all your *Unity* and C# projects clearly and organize them appropriately in well marked folders. Fellow team members and future programmers joining the team at a later stage will appreciate a well organized directory structure, and this will prevent important files from accidentally being deleted or getting lost.

A good directory structure is also essential to keep your game folder clean and tidy. Place all script, image, audio, font etc files in separate folders. With regards to *Unity*, create a well structured folder system in the *Asset* folder.

Note : Two game templates have been created for easier project creation. They can be found in the Dropbox folder.

- ***Backup*** : Additionally, always remember to backup your projects.