

Unity Coding Standards

Introduction

Establishing a coding standard is essential to the long-term health of a project. While there are certain general rules that are almost universally good to follow, other rules can be a matter of personal preference.

However, ***the most important thing is that all of these rules are followed***, once established.

Over time, the codebase will develop a sort of “second language” that will allow current engineers to read and understand new code easily, and new engineers to ramp-up and familiarize themselves with the codebase quickly.

It may seem like some rules put up some annoying hoops for you to jump through just to get a simple feature working. That’s good. These hoops may seem arduous, but they will force you to write good code.

Good code is:

- Easily readable and understandable, especially to other engineers who have not written it
- Flexible, and easy to modify or add to
- Sufficiently optimized for the needs of the project
- Written to expose appropriate properties for non-developers to tinker with in a way that makes sense (in Unity)

Good code is NOT:

- Written with a motivation to have as few lines and characters as possible
- Over-optimized to the point where the code becomes over-engineered and difficult to parse for non-authors
- Covered in comments explaining all aspects of it in detail. Reading the code should be self-explanatory (if written properly) and comments should only be used to explain parts of code that necessitate the use of more advanced/confusing practices.

With all of this being said, here are the rules that have been established for Unity projects as of January 2024 (projects started before this date may not fully adhere to them).

Code Formatting

Indentation

Rule: Code should be indented with 4 spaces, not tabs. If you're not a fan of mashing the spacebar (understandably), you can configure Visual Studio to automatically replace any tabs you have with the designated number of spaces instead.

Motivation: This is mostly personal preference, however in my experience using spaces tends to make files and diffs easier to read on Git.

Example:

Good Coding Example: Indentation



Bad Coding Example: Indentation



Spaces

Rule: Each distinct component of a line of code (variable name, expression character, keyword, etc.) should be separated by a space.

Motivation: Spreading out elements of code with a single space can make it easier for engineers to parse code without it feeling claustrophobic (this is admittedly a personal preference).

Example:

Good Coding Example: Spacing



Bad Coding Example: Spacing

Brackets

Rule: All brackets should be on a newline, and should not be put on the same line as a class name, conditional statement, etc.

Motivation: Placing brackets on newlines makes it easier for developers parse code that is particularly nest-y. Compartmentalizing code based on its indentation is easier to do when the brackets are essentially used as “markers” for this separation.

Example:

Good Coding Example: Brackets



Bad Coding Example: Brackets

Loops and If Statements

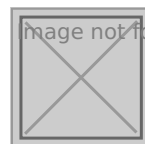
Rule 1: All loops and if statements, even if they execute a single line of code, should be enclosed in brackets.

Motivation: There is no functional purpose for writing single-line loops or if statements without brackets other than to save a trivial amount of characters and lines. Instead, it makes it more difficult to compartmentalize code while reading it, and makes it easier to accidentally break if another engineer tries to modify it (and forgets to add brackets that they assumed would be there).

Example:

Good Coding Example: Loops and Ifs Statements

Bad Coding Example: Else If Statements



Else If Statements

Rule: Else If and Else statements should be started on a newline, and should not be defined on the same line of a closing bracket.

Motivation: Formatting statements like this makes it easier for engineers to parse and mentally compartmentalize code, which is worth sacrificing a trivial number of extra lines.

Example:

Good Coding Example: Else If Statements

Bad Coding Example: Else If Statements



image not found or type unknown

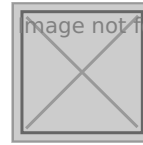


image not found or type unknown

Compound Conditional Statements

Rule: If a loop or if statement evaluates more than one condition at once, enclose those conditions with parentheses.

Motivation: Mostly personal preference. However, enclosing each condition in a compound statement makes it easier for engineers to determine which values are involved with which statement at a glance, especially for more complicated compound statements.

Example:

Good Coding Example: Compound Conditional Statements

Bad Coding Example: Compound Conditional Statements

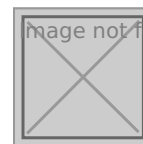


image not found or type unknown

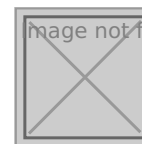


image not found or type unknown

Private Values

Rule: By default, C# designates all variables and functions as “private” unless designated otherwise by a keyword such as “public” or “protected.” However, we will be explicitly designating our values using the “private” keyword whenever it is applicable.

Motivation: This is once again a matter of personal preference. However, using the “private” keyword often results in code “lining up better” (particularly with lists of variables) and the color-coding that is

applied to most development environments can make these values easier to find, all of which results in “easier to read” code.

Example:

Good Coding Example: Private Values

Bad Coding Example: Private Values

image not found or type unknown

image not found or type unknown

Serialized Variables

Rule: When exposing a variable in Unity using the [SerializeField] Attribute, put it on a newline above the desired variable.

Motivation: This can make lists of variables easier to parse, as it keeps all of the variable declarations at the same indentation. It also makes it easier to see which variables are exposed in the Unity Inspector at a glance.

Example:

Good Coding Example: Serialized Variables

Bad Coding Example: Serialized Variables

image not found or type unknown

image not found or type unknown

Code/Naming Conventions

Loops and If Statements

Rule: Loops and conditional statements should be defined as explicitly as possible using two-sided expressions such as ==, >=, and <=. Avoid using != when possible, as well as evaluating a Boolean or other values by simply negating it with a !.

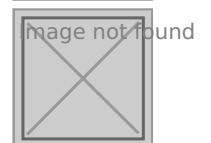
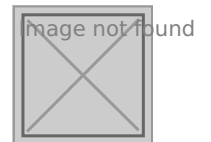
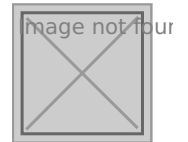
Motivation: Standardizing the way loops and conditional statements are defined lowers the risk of other engineers misinterpreting the condition. It also makes code easier to follow, because the purpose of the condition is explicitly described, and the data types of the variables involved are easy to identify.

Example:

Good Code/Naming Convention Example: Loops and If Statements

Bad Code/Naming Convention Example: Loops and If Statements

Bad Code/Naming Convention Example: Loops and If Statements



Variables

Ideally, if an engineer is looking over code written by someone else, they should be able to immediately comprehend what each variable represents, as well as its scope within the context of the code they see it in. With that in mind, we should strive to adhere to the following conventions when declaring and referencing variables.

Naming

Variable names should describe as accurately as possible the value it contains, regardless of how long it is. Ideally, an engineer will be able to understand what the variable is used for by simply reading the name, and not having to search for references of it to see how it's used in context.

Avoid abbreviating any words in a variable name, especially when it is not clear what word the abbreviation is meant to replace. The only exception to this rule is for variables that are declared solely to be incrementors in a for loop (ex: **for (int i = 0; i < 10; i++)**)

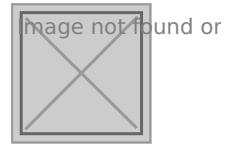


Code/Naming Convention Examples: Variables

Public Class Variables

Variables declared as public with a class-wide scope should be written in camelCase.

When referencing these variables within their declared class, they should be prefaced using the “this.” identifier to indicate their class-wide scope.



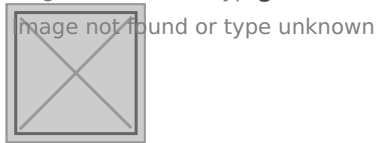
Good Code/Naming Convention Example: Public Class Variables

Protected Class Variables

Variables declared as protected with a class-wide scope should be written in camelCase, then ended with an underscore.

When referencing these variables, they should be prefaced using the “this.” identifier, to indicate their class-wide scope.

Good Code/Naming Convention Example: Protected Class Variables

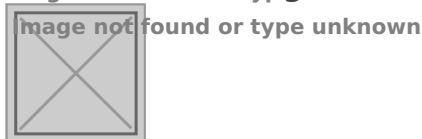


Private Class Variables

Variables declared as private with a class-wide scope should be prefaced by an underscore, then written in camelCase.

When referencing these variables, they should be prefaced using the “this.” identifier, to indicate their class-wide scope.

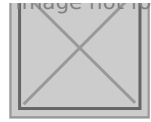
Good Code/Naming Convention Example: Private Class Variables



Function Parameters

Function parameters should be prefaced by an underscore, then written in camelCase.

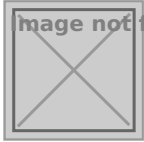
Good Code/Naming Convention Example: Function Parameters



Function Variables

Local variables declared within a function should be written in camelCase.

Good Code/Naming Convention Example: Function Variables



Static Variables

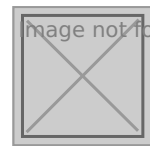
Variables declared as static should have each word in its name capitalized.

When referencing static variables, always preface the variable name with the name of the owning class

(even when you reference the static variable within the owning class).



Good Code/Naming Convention Example: Static Variables



Bad Code/Naming Convention Example: Static Variables

Constant Variables

Constant variables should be written in ALL CAPS, with underscores separating each word in the name.

When referencing constant variables, follow the same pattern as private, public, and static variables with

regards to prefacing identifiers.



Good Code/Naming Convention Example: Constant Variables

Functions

Function names should describe its purpose as accurately as possible. Ideally, an engineer should be

able to know what a function is supposed to do within a block of logic without having to look at its definition. With this in mind, functions should be declared with the following conventions:

- Each word in a function name should be capitalized.
- Avoid using abbreviations in a function name. Long descriptive names are better than short, abbreviated names.
- Whenever a function is referenced within the scope it was declared in, it should be prefaced with the “this.” identifier.
- Whenever possible, function names should start with a verb that describes what the function “does” (Ex: Get, Set, Access, Calculate, etc.)
 - If you find it difficult to name a function with a single verb, check to see if the function can actually be split into multiple smaller functions.

Classes

Class names should describe its purpose as accurately as possible. Ideally, an engineer should be able to understand the purpose of a class and have a relatively good idea of the functionality it contains by simply reading its name. With this in mind, classes should be declared using the following conventions:

- Each word in a class name should be capitalized.
- Avoid using abbreviations in a class name. Long descriptive names are better than short, abbreviated names.

Misc. Suggestions

The following are more subjective suggestions rather than hard-and-fast rules. However, you will likely find that following them will result in code that is cleaner and easier to read and understand at a glance.

Compartmentalize Complex Logic in a Function

If you are writing an algorithm (or part of an algorithm) that is several lines of code long, consider putting all of the specific logic into its own (properly named) function. Even if the function is only ever called once, it will be extremely helpful for other engineers (or yourself, in the future), as they will not be unnecessarily forced to parse and understand how a complex process works if they don’t have to.

Instead, they can read a descriptive function name, trust that it does what it says, and move onto other

parts of the code that they actually intend to investigate or modify.

Make Use of the [HideInInspector] Attribute

If a public variable is not meant to be set or modified in the Unity inspector, denote it with the [HideInInspector] attribute before its declaration. This will not only make components in the Unity editor far less cluttered, but it will also make it explicitly clear to other team members tinkering with the project which values are “allowed” to be experimented with without causing any technical issues.

Don’t Be Afraid to Copy

When in doubt about how parts of code should be formatted or named, it’s best to emulate what you already see in the codebase. In the end, it’s better to have a consistent coding style across the project than it is to have a “correct” style (a futile endeavor to pursue). Alternatively, you can simply ask another engineer on the team for advice on naming/formatting.

Revision #1

Created 17 December 2024 23:04:50 by Jesse Ferraro

Updated 17 December 2024 23:05:54 by Jesse Ferraro