Coding Guidelines | UNITY3D (GAMES)

**Software And Tools Version**

Use latest stable version of UNITY3D

Use and remain up to date with third party plugins

**Design Pattern And Framework**

Use MVC framework, most commonly used framework in software development UNITY3D use component based architecture, Use whenever is required.

Use pure Object Oriented Concept.

Follow single responsibility principle, i.e classes should only have one responsibility. Use modularisation

**File Header Template**

/\*

* Copyright 2016-2017 Appster Information Pvt. Ltd.
* All rights reserved.
* File: filename.h
* CodeLibrary/Project: CodeLibraryName/ProjectName
* Author: Author Name
* CreatedOn: date (dd/mm/yyyy)

\*/

**Naming**

Use meaningful, descriptive, consistent naming conventions for classes, interfaces, variables and methods.

Use Pascal case for class (Model, Controller, Request, Helper, Service etc.), properties, method and interface names. Use lower camel-case for variables.

Prefix m\_ for class member variables to differentiate class level and method level variables Use Pascal case for enum with e prefix.

Use Pascal case for file names. Use Pascal case for constants. Check spellings properly.

**Code Readability And Documentation**

Indent code and use white space for readability. The function code should not exceed 200 lines. The class code should not exceed NNN lines.

Comment your code to define the purpose of every class, method and parameter. Comment your code to explain logic behind a complex implementation.

Use XML tags comments before user-defined types such as a class, struct, or interface; members such as a field, event, property, or method; or a namespace declaration. XML tags comments help to generate web report.

Grouping class member using #region can help readability

**UNITY3D Best Practices and Optimization**

Cache component lookups: Whenever you access a component through GetComponent or an accessor variable, Unity has to find the right component from the game object. This time can easily be saved by caching a reference to the component in a private variable. Always Use built in array for fixed size. Instead of Array/ArrayList

Beware the empty Update functions. When creating new scripts with the Assets menu, they include an empty Update() function. Remove it if you don't need it.

Avoid Memory allocation. Every time an object is created, memory is allocated. Very often in code, you are creating objects without even knowing it. Example : Debug.Log(“Hello” + “World”);

Use Object Pooling instead of Instantiate and Destroy. You may have made games in the past that created and destroyed a lot of items (such as bullets or enemies). What you may not have known is that the act of instantiating and destroying are inefficient and can slow your projects down. So using object pooling for ephemeral objects is faster than creating and destroying them, because it makes memory allocation simpler and removes dynamic memory allocation overhead and Garbage Collection, or GC. Keep your FixedUpdate functions as lean as possible.

Refer to a GameObject by the most logical component. If you want to deal with an object's Transform, refer to it.

**Date and time**

Use the DateTime class.

**Errors And Exceptions**

Check NULL reference when use GameObject.Find

**Good coding practice**

Don't Repeat Yourself.

If you have multiple similar code blocks, put them in a function.

If you have multiple similar methods in different classes, put them in a superclass. Do not define helper methods in a model or controller. Put them in a helper class. Keep Functions Outside of Loops.

Remove extra use statements, variables, methods, classes etc. Try to Prevent Deep Nesting.

Put model related constants in the model class, not in config. e.g. User model can have constants/enum related to user status. Don't hard code any values. e.g. error messages etc. Move them to related config or resource files.

**Unit Testing:**

Test your code.

Make sure that you cover 70-80% of code using test cases.

This will reduce bugs in QA and reduce the overall effort in project development.

Coding Guidelines - JQuery

**Naming Convention**

Variable and function names should be full words, using camel case with a lowercase first letter. Names should be descriptive but not excessively so. Exceptions are allowed for iterators, such as the use of i to represent the index in a loop. Constructors do not need a capital first letter.

**Global Variables**

Each project may expose at most one global variable.

**Quotes**

jQuery uses double quotes.

*var double = "I am wrapped in double quotes";*

Strings that require inner quoting must use double outside and single inside.

*var html = "<div id='my-id'></div>";*

**Comments**

Comments are always preceded by a blank line. Comments start with a capital first letter, but don't require a period at the end, unless you're writing full sentences. There must be a single space between the comment token and the comment text. Single line comments go over the line they refer to:

/ *We need an explicit "bar", because later in the code foo is checked. var foo = "bar";*

/ *Even long comments that span*

/ *multiple lines use the single*

/ *line comment form.*



Multi-line comments are only used for file and function headers.

Inline comments are allowed as an exception when used to annotate special arguments in formal parameter lists or when needed to support a specific development tool:

*function foo( types, selector, data, fn, /\* INTERNAL \*/ one ) {*

*// Do stuff*

*}*

**Spacing**

In general, the jQuery style guide encourages liberal spacing for improved human readability. The minification process creates a file that is optimized for browsers to read and process.

Indentation with tabs.

No whitespace at the end of line or on blank lines.

Lines should be no longer than 80 characters, and must not exceed 100 (counting tabs as 4 spaces). There are 2 exceptions, both allowing the line to exceed 100 characters:

If the line contains a comment with a long URL.

If the line contains a regex literal. This prevents having to use the regex constructor which requires otherwise unnecessary string escaping.

if/else/for/while/try always have braces and always go on multiple lines.

Unary special-character operators (e.g., !, ++) must not have space next to their operand. Any , and ; must not have preceding space.

Any ; used as a statement terminator must be at the end of the line.

Any : after a property name in an object definition must not have preceding space. The ? and : in a ternary conditional must have space on both sides.

No filler spaces in empty constructs (e.g., {}, [], fn()) New line at the end of each file.

If the entire file is wrapped in a closure, the function body is not indented.

**Loading jQuery**

Always try to use a CDN to include jQuery on your page. Popular jQuery CDNs

Google: https://developers.google.com/speed/libraries/devguide Microsoft: <http://www.asp.net/ajaxlibrary/cdn.ashx>

jQuery: <http://jquery.com/download/> jsDelivr: [http://www.jsdelivr.com](http://www.jsdelivr.com/) cdnjs: [http://cdnjs.com](http://cdnjs.com/)

OSSCdn: [http://osscdn.com](http://osscdn.com/)

Keep all your JavaScript and jQuery includes at the bottom of your page.

**What version to use?**

DO NOT use jQuery version 2.x if you support Internet Explorer 6/7/8.

For new web-apps, if you do not have any plugin compatibility issue, it's highly recommended to use the latest jQuery version. When loading jQuery from CDN's, always specify the complete version number you want to load (Example: 1.11.0 as opposed t

1.11 or just 1).

DO NOT load multiple jQuery versions.

DO NOT use jquery-latest.js from jQuery CDN.

For advanced browser feature detection, use Modernizr .

**jQuery Variables**

All variables that are used to store/cache jQuery objects should have a name prefixed with a $. Always cache your jQuery selector returned objects in variables for reuse.

var $myDiv = $("#myDiv");

$myDiv.click(function(){...}); Use camel case for naming variables.

**Selector**

Use ID selector whenever possible. It is faster because they are handled using document.getElementById(). When using class selectors, don't use the element type in your selector.

*var $products = $("div.products"); // SLOW var $products = $(".products"); // FAST*

Use find for Id->Child nested selectors. The .find() approach is faster because the first selection is handled without going through the Sizzle selector engine.

/ *BAD, a nested query for Sizzle selector engine var $productIds = $("#products div.id");*

/ *GOOD, #products is already selected by document.getElementById() so only div.id needs to go through Sizzle selector engine*

*var $productIds = $("#products").find("div.id");*

Be specific on the right-hand side of your selector, and less specific on the left.

*// Unoptimized*

*$("div.data .gonzalez");*

*// Optimized*

*$(".data td.gonzalez");*

Avoid Excessive Specificity.

*$(".data table.attendees td.gonzalez");*

/ *Better: Drop the middle if possible. $(".data td.gonzalez");*

Give your Selectors a Context.

/ *SLOWER because it has to traverse the whole DOM for .class $('.class');*

/ *FASTER because now it only looks under class-container.*

*$('.class', '#class-container');*

Avoid Universal Selectors.

*$('div.container > \*'); // BAD*

*$('div.container').children(); // BETTER*

Avoid Implied Universal Selectors. When you leave off the selector, the universal selector (\*) is still implied.

*$('div.someclass :radio'); // BAD*

*$('div.someclass input:radio'); // GOOD*

Don’t Descend Multiple IDs or nest when selecting an ID. ID-only selections are handled using document.getElementById() so don't mix them with other selectors.

*$('#outer #inner'); // BAD*

*$('div#inner'); // BAD*

*$('.outer-container #inner'); // BAD*

*$('#inner'); // GOOD, only calls document.getElementById()*

**DOM Manipulation**

Use string concatenation or array.join() over .append()

*// BAD*

*var $myList = $("#list"); for(var i = 0; i < 10000; i++){*

*$myList.append("<li>"+i+"</li>");*

*}*

*// GOOD*

*var $myList = $("#list"); var list = "";*

*for(var i = 0; i < 10000; i++){*

*list += "<li>"+i+"</li>";*

*}*

*$myList.html(list);*

/ *EVEN FASTER*

*var array = [];*

*for(var i = 0; i < 10000; i++){*

*array[i] = "<li>"+i+"</li>";*

*}*

*$myList.html(array.join(''));*

Don’t Act on Absent Elements.

/ *BAD: This runs three functions before it realizes there's nothing in the selection $("#nosuchthing").slideUp();*

/ *GOOD*

*var $mySelection = $("#nosuchthing"); if ($mySelection.length) {*

*$mySelection.slideUp();*

*}*

**Events**







Use only one Document Ready handler per page. It makes it easier to debug and keep track of the behavior flow.

DO NOT use anonymous functions to attach events. Anonymous functions are difficult to debug, maintain, test, or reuse.

*// BAD*

*$("#myLink").on("click", function(){...});*

*// GOOD*

*function myLinkClickHandler(){...}*

*$("#myLink").on("click", myLinkClickHandler);*

Document ready event handler should not be an anonymous function. Once again, anonymous functions are difficult to debug, maintain, test, or reuse.

*$(function(){ ... }); // BAD: You can never reuse or write a test for this function.*

*// GOOD*

*$(initPage); // or $(document).ready(initPage);*

*function initPage(){*

*// Page load event where you can initialize values and call other initializers.*

*}*

Document ready event handlers should be included from external files and inline JavaScript can be used to call the ready handle after any initial setup.

*<script src="my-document-ready.js"></script>*

*<script>*

*// Any global variable set-up that might be needed.*

*$(document).ready(initPage); // or $(initPage);*

*</script>*

DO NOT use behavioral markup in HTML (JavaScript inlining), these are debugging nightmares. Always bind events with jQuery to be consistent so it's easier to attach and remove events dynamically.

*<a id="myLink" href="#" onclick="myEventHandler();">my link</a> <!-- BAD*

*$("#myLink").on("click", myEventHandler); // GOOD*

When possible, use custom namespace for events. It's easier to unbind the exact event that you attached without affecting other events bound to the DOM element.

*$("#myLink").on("click.mySpecialClick", myEventHandler); // GOOD*

/ *Later on, it's easier to unbind just your click event $("#myLink").unbind("click.mySpecialClick");*

**Ajax**







Avoid using .getJson() or .get(), simply use the $.ajax() as that's what gets called internally.

DO NOT use http requests on https sites. Prefer schemaless URLs (leave the protocol http/https out of your URL) DO NOT put request parameters in the URL, send them using data object setting.

/ *Less readable...*

*$.ajax({*

*url: "something.php?param1=test1&param2=test2",*

*....*

*});*

/ *More readable...*

*$.ajax({*

*url: "something.php",*

*data: { param1: test1, param2: test2 }*

*});*

Try to specify the dataType setting so it's easier to know what kind of data you are working with. Use Promise interface:

*$.ajax({ ... }).then(successHandler, failureHandler);*

*// OR*

*var jqxhr = $.ajax({ ... }); jqxhr.done(successHandler); jqxhr.fail(failureHandler);*

Chaining

Use chaining as an alternative to variable caching and multiple selector calls.

 *$("#myDiv").addClass("error").show();*

Whenever the chain grows over 3 links or gets complicated because of event assignment, use appropriate line breaks and indentation to make the code readable.

*$("#myLink")*

*.addClass("bold")*

*.on("click", myClickHandler)*

*.on("mouseover", myMouseOverHandler)*

*.show();*

For long chains it is acceptable to cache intermediate objects in a variable.

**Miscellaneous**

Use Object literals for parameters.

*$myLink.attr("href", "#").attr("title", "my link").attr("rel", "external"); // BAD, 3 calls to attr()*

/ *GOOD, only 1 call to attr() $myLink.attr({*

*href: "#",*

*title: "my link", rel: "external"*

*});*

Do not mix CSS with jQuery.

*$("#mydiv").css({'color':red, 'font-weight':'bold'}); // BAD*

*.error { color: red; font-weight: bold; } /\* GOOD \*/*

*$("#mydiv").addClass("error"); // GOOD*