# Contribution Guidelines

Thank you for visiting our contribution guidelines! An active and healthy development community is what makes a good game engine an exceptional game engine. As we focus on developing new features and resolving bugs with every version of Lumberyard, we want to hear from you. We are interested in seeing how you're using the engine and what improvements you're making while you work on your own game projects. This is why, in addition to our [GameDev Forums](https://gamedev.amazon.com/forums/index.html), [Tutorials](https://www.youtube.com/amazongamedev) and [Documentation](https://aws.amazon.com/documentation/lumberyard/), we provide you with the opportunity to share your features and improvements with your fellow developers. After you modify the core engine code, simply submit a pull request.

To make it easy for you to contribute to our game engine, the Lumberyard development team adheres to the following coding conventions. We believe that these guidelines keep the engine code consistent and easy to understand so that you can spend less time interpreting code and more time coding. We look forward to your contributions!

## Compiler Compatibility:

- Use the C++11 standard whenever possible.

- Stick to the C++11 features that are commonly supported by Microsoft Visual Studio 2013/2015 (refer to https://msdn.microsoft.com/en-us/library/hh567368.aspx).

## Formatting:

- Lumberyard recommends using the Uncrustify code beautifier to keep C++ code consistent with the engine code. Refer to http://uncrustify.sourceforge.net/.

- Apply indentation in a consistent manner:

- Files should start without any indentation.

- Use a single additional level of indentation for each nested block of code.

- Indent all lines of a block by the same amount.

- Make lines a reasonable length.

- Indent preprocessor statements in a similar way to regular code.

- When positioning curly braces, open braces on a new line and keep them flush with the outer block's indentation.

- Always use curly braces for flow control statements.

- Each line of code should only include a single statement.

- Naming conventions for classes, functions, types and files should adhere to CamelCase and specify what the function does.

- All header files must include the directive, "#pragma once".

- Use forward declarations to minimize header file dependencies. Compile times are a concern so please put in the effort to minimize include chains.

- The following syntax should be used when including header files: #include <Package/SubdirectoryChain/Header.h>

This rule helps disambiguate files from different packages that have the same name. <Object.h> might appear relatively often, but <AZRender/Object.h> is far less likely to.

## Classes:

- You should define a default constructor if your class defines member variables and has no other constructors. Unless you have a very specifically targeted optimization, you should initialize all variables to a known state even if the variable state is invalid.

- Do not assume any specific properties based on the choice of struct vs class; always use <type\_traits> to check the actual properties

- Public declarations come before private declarations. Methods should be declared before data members.

- All methods that do not modify internal state should be const. All function parameters passed by pointer or reference should be marked const unless they are output parameters.

- Use the override keyword wherever possible and omit the keyword virtual when using override.

- Use the final keyword where its use can be justified.

## Scoping:

- All of your code should be in at least a namespace named after the package and conform to the naming convention specified earlier in this document.

- Place a function's variable declarations in the narrowest possible scope and always initialize variables in their declaration.

- Static member or global variables that are concrete class objects are completely forbidden. If you must have a global object it should be a pointer, and it must be constructed and destroyed via appropriate functions.

## Commenting Code:

Clear and concise communication is essential in keeping the code readable for everyone. Since comments are the main method for communication, please follow these guidelines for commenting the code:

- Use /// for comments.

- Use /\*\*..\*/ for block comments.

- Use @param, etc. for commands.

- Full sentences with good grammar are preferable to abbreviated notes.