**Game Studio Project   
Coding Standards and Conventions  
Draft 1**

**Filename Conventions**

All filenames should use PascalCase.

FileName.cs

**Implementation and Interface**

Interface of a class should be separate from its implementation.

// Define the interface (in one file)

namespace Namespace

{

public interface Interface;

{

void Method();

}

}

// Implement the interface in a class (in another file)

namespace Namespace

{

public class Class : Interface

{

public void Method()

{

//Method implementation

}

}

}

// Instantiate class instance in code

...

if (true)

{

Interface object = new Class();

object.Method();

}

...

**Indented code inside new scope**

All scopes like if, else, etc. The only exception should be getters and setters. Braces should be alone on their respective lines and are always required regardless of whether they are needed.

{

//Indent

{

//Indent

}

}

**Variable naming convention**

Variables should be named using camelCase. Member attributes should be prefixed with “m\_”. Static attributes should be prefaced with “s\_”. All parameters should be prefaced with “\_”.

...

//Variable

exampleVariable;

//Member attribute

m\_exampleAttribute;

//Static attribute

s\_exampleAttribute;

//Function parameter

\_exampleParameter;

...

**Function, class, struct, interface, namespace naming conventions**

Functions, classes, structs, interfaces and namespaces should be named using PascalCase.

...

//Indent

ExampleClass;

...

**Global variables**

Global variables should not be used.

**Magic numbers**

Magic numbers should be defined in the appropriate scope; in the scope they are required, or either in a parent scope if used frequently, or if they should be centrally managed (like a setting), defined as a static attribute.

**Meaningful names for iterators**

Loops should use meaningful names for iteratative variables.

//For loop

foreach (int num in nums) //As opposed to foreach (int i in nums)

{

...

}

**Complex conditions or parameter lists should be separated over lines**

if (

a == 1 &&

b == 2 &&

c == 3

)

{

...

}

**Classes**

Classes should explicitly declare public/private variables/methods. Classes should encapsulate (privatise) all member attributes. Classes should expose them using getters and setters. Getters and setters should be formatted as such (for readability):

class Class

{

private int \_number;

public int m\_number

{

get {return \_number;}

set {\_number = value;}

}

}

If a getter/setter needs validation, treat it like every other scope (separate out the curly braces and indent the codeblock):

class Class

{

private int m\_amount = 0;

public int GetAmount()

{

//Some logic

return m\_amount;

}

public void SetAmount(int \_amount)

{

//Some logic

m\_amount = \_amount;

}

}

**Using**

Use throwaway objects to immediately require the garbage collector to dispose them out of memory, if the object is resource intensive.

using (var exampleObject = new ExampleClass("example.txt"))

{

// Use myObject

}

// myObject is automatically disposed of here

**Exceptions (Unity)**

Gracefully handle potential exceptions using try/catch blocks and Unity’s exception logging.

try

{

// Your code that might throw an exception

}

catch (Exception ex)

{

Debug.LogException(ex, this);

}

**Switch cases and extended if/else conditional blocks**

A map should be used instead of these structures (performance consideration).

**Inlining/optimising functions**

Inline single line functions or very regularly-called functions (like pushing/popping queues). Optimise heavily algorithmic functions to make them more performant (enemy ai, pathfinding, etc).

public class Example

{

[MethodImpl(MethodImplOptions.AggressiveInlining)]

public void InlinedMethod()

{

// Code that should be inlined

}

[MethodImpl(MethodImplOptions.AggressiveOptimization)]

public void OptimizedMethod()

{

// Code that should be aggressively optimized

}

}

**Lazy initialisation**

Consider lazy initialisation of objects for large data structures.

...

//Initialisation

private MyClass myObject = new MyClass();

//Lazy initialisation

private Lazy<MyClass> myObject = new Lazy<MyClass>(() => new MyClass());

...

**XML Documentation**

Use XML documentation coding standards (to be run through Doxygen to produce documentation at project finalisation).

/// <summary>

/// Calculates the sum of two integers.

/// </summary>

/// <param name="a">The first integer.</param>

/// <param name="b">The second integer.</param>

/// <returns>The sum of the two integers.</returns>

public int Add(int a, int b)

{

return a + b;

}