Moai Lua Style Guide

Version 1.0
January 7, 2013

Indentation

Use tabs, not spaces. Set tabs to display as four spaces in your text editor.

Extra Spaces

Include spaces after brackets and parentheses. Omit spaces between groups of brackets or parentheses:

```
nI = (( nA + nB ) * nC ) - tD [ 19 ];
tEnvironment = setmetatable ({}, { __index = _G })
function foo ( nA, nB, nC ) return nA + nB + nC end
```

Add spaces on both sides of binary operators and to the left of the unary minus and length operators (but not the right):

```
nA = nA + -nB
bFlag = not bFlag
nLength = #tTable
```

Dividers

The ‘thick’ divider looks like this:

```
--================================================================--
-- Title
--================================================================--
```

The ‘thin’ divider looks like this:

```
----------------------------------------------------------------
```

Section specific comments always go below the divider, never above:

```
--================================================================--
-- Title
--================================================================--
-- Comments about this section

----------------------------------------------------------------
-- Comments about this section
```

The ‘dashed’ divider looks like this:

```
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
```

Use the ‘dashed’ divider as to highlight blocks of code, groups of table members, nested functions, etc.:

```
----------------------------------------------------------------
function veryLongFunction ()

    -- <lots of code here>

    -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
    local function someNestedFunction ()
    end

    -- <more code here>
```

```
end
```

String Constants

Prefer single quotes to double quotes when declaring string constants:

```
-- Like this
sString = ‘this is a string constant’

-- Not this
sString = “this is a string constant”
```

Table Declarations

Place the opening brace of any non-inline table declaration on the same line of the statement. Indent the contents of the declaration by a single level. Align the closing brace with the statement. Leave a comma before any newline.

```
// This is OK
local tFoo = {
   nX = 1,
   nY = 2,
   nX = 3,
}

// Mostly avoid doing this
local tBar = { nCount = 0 }

// Don’t do this
local tBax = { nX = 1, nY = 2, nZ = 3 }
```

Variables

Always use TitleCase prefixed by a single lowercase letter to indicate the intended use of the variable or parameter:

```
nCount   = 0 -- number
bDone    = false -- bool
sName    = ‘Moai’ -- string
```

For tables, indicate if the table is to be used as a numbered array, metatable or general purpose table:

```
aNumberedArray    = { foo, bar, baz }
mMetatable        = { __index = _G }
tDictionary       = { foo = 1, bar = ‘baz’ }
```

For variables or parameters that may be assigned multiple types, use the prefix ‘v’:

```
function acceptNumberOrString ( vParam )
```

Alignment

When possible, align groups of assignments using tabs:

```
nFoo        = 1
nFooBar     = 2
nFooBarBaz  = 3

tTable = {
    nFoo        = 1,
```

```
    nFooBar     = 2,
    nFooBarBaz  = 3,
}
```

Function Naming and Declarations

If a function variable is intended to be the ‘proper name’ of a function (or is a class member) then use camelCase. If a function variable is intended to be assigned a value (is a user-assignable class member or callback parameter, for example) then use TitleCase prefixed with the lowercase letter ‘f’:

```
-- Declare functions this way
function someFunc ( fCallback ) -- function that takes a callback

-- Don’t do this
someFunc = function ( fCallback )

-- Declare member functions like this
function tSomeTable.someFunc ()

-- If a member function uses ‘self’ declare it like this
function tSomeTable:someFunc ()
```

Globals

For global and file-scope variables that aren’t classes use ALL_CAPS prefixed by a single letter type indicator:

```
N_TOTAL_PLAYERS      = 0 -- number
T_INVENTORY          = {} -- table
B_HAS_CONNECTION     = false -- bool
S_USER_NAME          = ‘Moai’ -- string
```

If the global is a function variable that is intended to be assigned to:

```
F_SELECTOR_FUNC      = function () end -- function
```

Consts

For consts that aren’t functions or classes, use ALL_CAPS prefixed by CONST and a single letter type indicator:

```
CONST_N_MAX_CONNECTIONS    = 5 -- number
CONST_T_LOCALIZED_TEXT     = {} -- table
CONST_B_DEBUG              = false -- bool
CONST_S_SERVICE_URL        = ‘https://services.server.com’ -- string
```

Classes

Name classes using TitleCase. Use a ‘thick’ divider above the class creation. Follow this by any public, private or static members then by the class methods, alphabetized by name. Place a ‘thin’ divider before each method. Do not inline method definitions in the scope tables. Call the ‘prepare’ method at the end of the definition.

```
--============================================================--
-- ExampleClass
--============================================================--
Class.makeSubclass ( ‘ExampleClass’ )
```

```
ExampleClass.private = {
    nX = 0,
    nY = 0,
    nZ = 0,
}

ExampleClass.public = {
    nI = 0,
    nJ = 0,
    nK = 0,
}

ExampleClass.static = {
    N_FOO = 0,
    N_BAR = 0,
    N_BAZ = 0,
}

-----------------------------------------------------------------
function ExampleClass.abstract.abstractMethod1 () end
function ExampleClass.abstract.abstractMethod2 () end
function ExampleClass.abstract.abstractMethod3 () end

-----------------------------------------------------------------
function ExampleClass.static.methodA ()
end

-----------------------------------------------------------------
function ExampleClass.public:methodB ()
end

-----------------------------------------------------------------
function ExampleClass.private:methodC ()
end

ExampleClass.prepare ()
```

Some keywords and special tables associated with the class system are exempt from usual variable naming rules, as are class instances:

```
-- scope tables
public
private
abstract
static


-- hierarchy and interface acessors
self
this
super

-- class instance
foo = Foo.new ()
```

Member Variables

Follow the usual variable naming rules to indicate type and usage:

```
T_GLOBAL = {
    sSomeString = ‘foo’
    nSomeNumber = 42
    CONST_N_MAX = 500
}
```

In the case of CONST_ tables, members should be ALL_CAPS, but the CONST_ prefix should be dropped:

```
CONST_T_GLOBAL = {
    N_SOME_STRING = 'foo'
    N_SOME_NUMBER = 42
    N_MAX = 500
}
```

Static class members should follow the same convention used for globals:

```
ExampleClass.static = {
    N_FOO        = 0,
    N_BAR        = 0,
    CONST_N_BAZ  = 0,
}
```

Class Name Postfixes

The following class name postfixes are reserved and have the following special meanings:

1. **Base:** An abstract base class or any class specifically intended to be inherited.
2. **Mgr:** A singleton object or a class containing only class methods meant to manage static state.
3. **Util:** A class containing only class methods and no state or a namespace containing only functions. For example 'StringUtil.'

Acronyms

Acronyms should always follow the case style of the name that includes them:

```
UfoClassName -- TitleCase
nUfoLocalVar -- camelCase
CONST_S_UFO_NAME -- ALL_CAPS
```

Acronyms should never contradict the case style of the name that includes them:

```
UFOClassName -- wrong
nUFOLocalVar -- do not
CONST_S_Ufo_NAME -- bad
```

Accessor Methods

Use verb prefix. The following verb prefixes are reserved for accessors:

1. **Get:** Returns a property.
2. **Is:** Boolean a Boolean property.
3. **Set:** Sets a property.

```
// This is OK
function Foo.public:getName ()
function Foo.public:isVisible ()
function Foo.public:setName ( sName )
function Foo.public:setVisible ( bVisible )
```

Method Name Verbs

1. **Affirm:** Lazy initialization. If an object or member doesn't exist, it will be created and initialized. If it already exists, nothing is done. In the context of a collection, adds an object to the collection only if object is not already in the collection. Affirm may or may not have a return value.
2. **Clear:** Releases resources associated with an object. If object is a container, removes all elements. Object should remain initialized and suitable for use after a call to a Clear method.
3. **Contains:** Reserved for collections. Boolean check to see if the set contains an object.
4. **Copy:** Allocates and initializes a new instance using an existing instance. May or may not make a 'deep' copy.
5. **Create:** Do not use this verb. Use 'New' instead.
6. **Init:** Initialize an object.
7. **Insert:** Add an object to a collection.
8. **New:** Returns a new instance of a class.
9. **Remove:** Remove an object from a collection.