Coding Standards

File Formatting

Indentation

Each level of code should be indented using four spaces and use of tab-spacing should be avoided.

Libraries

Library imports should be ordered alphabetically. The top level library should appear first when importing multiple sub libraries.

Example 1:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

Example 2:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;
```

Spacing

Developers should follow the below spacing guidelines:

- Place two blank lines after library imports
- Place two blank lines after the final curly brace of each class
- Place one blank line before each function declaration
- Place one blank line to separate public member variables from private member variables
- Place additional blank lines to separate distinct sections of code

Naming Conventions

Classes

Class names should utilize the PascalCase naming scheme where the first letter of each word is capitalized. There should be no special characters or numbers present in the class name. The name of a class should always match the name of the file it resides in. Enum and struct types should also follow this naming convention.

Example 1:

```
public class CustomClass
{
}
```

Functions

Function names should utilize the PascalCase naming scheme where the first letter of each word is capitalized. There should be no special characters or numbers present in the function name.

Example 1:

```
void CustomFunction()
{
}
```

Variables

Most variable names should follow the camelCase naming scheme where the first letter of each word is capitalized, excluding the first word. However, member variables should be preceded by the “m_” prefix.

Example 1:

```
public class Person
{
    string m_name;
    int m_age;

    bool StartsWithA()
    {
```

```
        char letterA = 'A';
        if (m_name[0] == letterA)
        {
            return true;
        }
        return false;
    }
}
```

Files

C# file names should always be identical to the class they contain. If a file contains multiple classes then the developer should ensure the file name matches the name of the most relevant class.

Names for any non-code Unity files should follow the PascalCase naming scheme where the first letter of each word is capitalized. There should be no special characters present in any file names. Numbers can be appended to file names to distinguish between versions or files with similar names.

Folders

Folders should follow the PascalCase naming scheme where the first letter of each word is capitalized. There should be no special characters present in any folder names.

Documentation

Files

File comments should be placed at the very beginning of the file before any library imports or code. They should follow the below format.

```
/*
 * Filename: CustomClass.cs
 * Developer: Your Name
 * Purpose: This file demonstrates comments.
 */
```

Classes

Class comments should be placed directly before the class declaration. The first line should be a brief summary and follow the format below. Omit the member variables section when no member variables are present.

```
/*
 * An empty class to demonstrate comments.
 *
 * Member variables:
 * m_isClass -- Boolean to demonstrate a member variable comment.
 */
public class CustomClass
{
    int m_isClass = true;
}
```

Functions

Function comments should be placed directly before the function declaration, following the format below. Comments are not necessary for Unity functions. Omit the parameters and returns section if the function does not take parameters or is void, respectively.

```
/*
 * Function to check if a name starts with the letter 'A'.
 *
 * Parameters:
 * name -- The name to check the first letter of.
 *
 * Returns:
 * bool -- True if the name starts with A, false otherwise.
 */
bool StartsWithA(string name)
{
    if (name[0] == 'A')
    {
        return true;
    }
    return false;
}
```

Additional Comments

Additional comments can be placed inline using the “//” comment style, as shown below.

```
// If statement for checking letter
if (m_name[0] == letterA)
```

```
{
    return true; // The first letter is an A.
}
```

Prefabs

Prefabs should be documented with a “.txt” file in the same directory as the prefab it describes. Text files should follow the below format.

```
Prefab: SuperCoolEnemy
Purpose: Represent the enemy and allow the application of a nav mesh.
Interactions: This prefab loses health when it collides with the player mesh collider.
```

Other Formatting

Files

The following general rules should be followed when formatting your code:

- Max line length should not exceed more than ~120 characters
    - Can be exceeded in cases of long Debug.Log() calls
- Curly braces should be placed on the line after function, class, or other declarations

Classes

- Public variables should appear before private variables in classes
- Public functions should appear before private functions in classes
- Unity functions should appear before custom functions

Error Handling

Exceptions

Exceptions should be used whenever there is the reasonable possibility of a major error occurring.Unity’s exception logging syntax should be used to print the error, as in the example below.

```
try
{
    // Area where an error may occur
}
catch (Exception e)
```

```
{
    Debug.LogException(e, this);
}
```

Test Cases

Stress tests should print a message indicating the conditions of the test at its completion. The message should be in the format “[filename.cs -- FunctionName()] Stress test ended: description of condition.” A shortened example is below.

```
[UnityTest]
public IEnumerator CustomStressTest()
{
    // Perform stress test

    if (fps < 30)
    {
        Debug.Log("[MyStressTest.cs -- CustomStressTest()] Stress test
ended: 30 FPS reached after " + numItems + " items were spawned.");
        break;
    }
}
```