Coding Standard

Goals

Efficiently develop code and applications with the following qualities :

- **Robust** : runs without crash and protects the data from being lost or corrupted.
- **Secure** : protects the data from being stolen or hacked.
- **Ergonomic** : does exactly what the user needs and can be used in an productive and intuitive manner.
- **Efficient** : minimizes processing times to maximize the user productivity.
- **Maintainable** : is easy to fix and enhance by any programmer in the team.
- **Extensible** : is easy to extend with new features by reusing existing components.
- **Consistent** : looks like it has been designed and implemented by a single developer.

Specificity

This coding standard favors readability over compactness, by :

- Forbidding the use of cryptic acronyms, abbreviations, prefixes and suffixes;
- Using different letter cases for classes, class members and local variables;
- Including the class name in the attribute and variable names.

Rules

- Develop the application and its components with simple, robust and efficient code which will be easy to understand, extend and debug by any programmer in the team.
- Develop any piece of code so that it's :
  - easy to understand just by itself;
  - impossible to guess who has actually worked on it.
- Use **American English** for all the code, including comments.

```
InitializeColor();
MoveForward();
```

- Use the **meter** as the default distance unit.
- Use the **second** as the default time unit.
- Use **four spaces** instead of tabulations, to make the code independent of the editor settings.
- Choose **short meaningful identifiers** for class, attribute, method, constant and variable names, to prevent ambiguity and cognitive load.
- Use **standard prefixes** :
  - First, Last, Post
  - Prior, Next
  - Sub, Super, Base

    - Initial, Final
    - Old, New
    - Backward, Forward
    - Left, Right
    - Back, Front
    - Bottom, Top
    - Minimum, Maximum
    - Lower, Higher, Upper
    - Horizontal, Vertical
    - Local, Global
- Use **standard suffixes** :
    - Index, Count
    - Array, List, Map, Dictionary
- Use **standard verbs** :
    - Initialize, Update, Finalize
    - Create, Release, Destroy
    - Build, Apply
    - Clear, Fill
    - Reset, Set, Get, Find
    - Is, Has
    - Add, Remove
    - AddFirst, RemoveFirst
    - AddLast, RemoveLast
    - Start, Stop
    - Begin, End
    - Enter, Exit
    - Open, Close
    - Read, Write
    - Load, Save
    - Pause, Resume
    - Lock, Unlock
    - Attach, Detach
    - Enable, Disable
    - Select, Deselect
    - Activate, Deactivate
    - Increment, Decrement
    - Increase, Decrease
    - Compress, Decompress
    - Connect, Disconnect
    - Send, Receive
    - Grant, Revoke
- Name your **types** (classes, structures, enumerations, etc) in **UPPER_CASE**, without articles.

```
class TANK_SHELL
{
    VECTOR_3
        PositionVector;
    QUATERNION
        RotationQuaternion;

    ...
}
```

- Name your **type members** (methods, attributes, constants, etc) in **PascalCase**, without articles.

```
ShootShell(
    Muzzle.PositionVector,
    Muzzle.RotationQuaternion
    );
```

- Name your **local variables** and **method parameters** in **snake_case**, without articles.

```
void ShootShell(
    VECTOR_3 shell_position_vector,
    QUATERNION shell_rotation_quaternion
    )
{
    TANK_SHELL
        shot_tank_shell;

    ...

    shot_tank_shell
        = new TANK_SHELL(
              shell_position_vector,
              shell_rotation_quaternion
              );

    ...
}
```

- Don't use abbreviations or single-letter variables.

```
TANK FindTank(
    int tank_identifier,
    int first_tank_index,
    int post_tank_index
    )
{
    int
        tank_index;

    for ( tank_index = first_tank_index;
```

```
            tank_index < post_tank_index;    // no i, j, n, etc
            ++tank_index )
    {
        if ( TankArray[ tank_index ].Identifier == tank_identifier )
        {
            return TankArray[ tank_index ];
        }
    }

    return null;
}
```

- Avoid acronyms, and capitalize them in member names.

```
DATABASE_URL
    DatabaseUrl;
```

- If a variable name collides with a predefined identifier, simply add a trailing underscore.

```
CLASS
    class_;

class_ = new CLASS;
```

- Use a noun or noun phrase for classes, constants, attributes and variable names.
- Include the meaningful part of the class name in attribute and variable names.

```
Dictionary<PLAYER, string>
    ActivePlayerDictionary;
List<ENEMY>
    CloseEnemyList;
TANK[]
    EnemyTankArray;
VECTOR_3
    InitialShellPositionVector,
    TankVelocityVector;

void ShootShell(
    )
{
    SHELL
        last_shot_shell,
        shot_shell;

    ...
}
```

- Start method names with a verb in the imperative mood (Set, Get, Find, ...).
- Start boolean inquiry names with a verb in the indicative mood (Is, Has, Can, ...).

- Declare the method parameters in the same order as in the method name.

```
bool FindPlayerIndexByName(
    ref int player_index,
    string player_name
    )
{
    ...
}
```

- Use a positive affirmation for boolean variable and attribute names.

```
if ( game_is_paused )
{
    ...
}
```

- If an attribute name starts like its owner class name, omit the common prefix.

```
class TANK
{
    TANK_SHELL[]
        ShellArray;
    bool
        IsDamaged;

    void ShootShell(
        TANK_SHELL tank_shell
        )
    {
        ...
    }
}
```

- Align matching braces.

```
bool CanShoot(
    )
{
    return ShotShellCount < MaximumShellCount;
}

// ~~

void ShootShell(
    VECTOR_3 initial_velocity_vector
    )
{
    ...
}
```

- Use braces for single statement blocks.

```
if ( LoadedAmmunitionCount > 0 )
{
    ShootBullet();
}
else if ( CarriedAmmunitionCount > 0 )
{
    ReloadWeapon();
}
else
{
    NoAmmunitionSound.Play();
}
```

- Declare each attribute, variable and method parameter name on separate line.

```
int
    tank_count,
    tank_index;
```

- Try to declare all local variables at the start of the method, to improve the algorithm readability.
- Group local variables of the same type, and sort the declarations by ascending types (lowercase, then PascalCase, then UPPER_CASE) and variable names, so that the declaration of a variable can be located at a glance.

```
int
    shell_count,
    shell_index,
    tank_count,
    tank_index;
string
    player_name,
    target_name;
CharacterController
    character_controller;
NavMeshAgent
    navigation_mesh_agent;
TANK
    enemy_tank;
TANK[]
    enemy_tank_array;
```

- Split complex statements and expressions on several lines if they contain boolean operators or if they become too wide.
- When splitting an expression on several lines, start the next lines with an operator and align it with the start of its left operand (or else indent it by 4 spaces).

```
if ( ( tower.GetDistance(
            tower_target,
            weapon_type
            )
       > tower.MaximumShootingDistance )
     || ( tank_distance > maximum distance
          && tank_health > 0.5 ) )
{
    ...
}
```

- Put the scalar or constant multiplier after the multiplicand expression.

```
average_value = ( first_value + second_value ) * 0.5f;
```

- Add exactly one space :
  - after `(` `[` `,`
  - before `)` `]`
  - after `if` `while` `for` `foreach` `return` ...
  - around operators
- Add exactly one empty line :
  - around standard comments;
  - after the local variable declarations;
  - after the method preconditions;
  - between `if` `while` `for` `foreach` `do` `return` and the prior statement;
  - between `}` and the next statement.
- Use standard file extensions.
  - C# : cs
  - C : c, h
  - C++ : cpp, hpp
  - Javascript : js
  - HTML : html
  - CSS : css
- Declare one class per source code file.
- Use the class name in lowercase as file name.

```
tank_shell.cpp
tank_shell.hpp
```

- Use the class name in uppercase as Unity file name.

```
TANK_SHELL.cs
```

- Group the class elements by category, declared in the same predefined order :

    - Imports.
    - Types.
    - Constants.
    - Attributes.
    - Constructors.
    - Destructor.
    - Operators.
    - Inquiries : methods which don't change the class attributes.
    - Operations : methods which change the class attributes.
- Within a category, declare :
    - the called methods before the calling methods, preferably in the order they will be called, so that the class code can be immediately understood by a single sequential read.
    - the static members after the non-static members.
- Import exactly what each file needs to be compiled independently, and nothing more.
- Sort the imports by ascending names.
- Use public attributes and methods, unless you need to declare them as private.
- Delimitate the code sections with standard comments.

```
// -- IMPORTS

...

// -- TYPES

class NAME
{
    // -- CONSTANTS

    ...

    // -- ATTRIBUTES

    ...

    // -- CONSTRUCTORS

    ...

    // -- DESTRUCTOR

    ...

    // -- OPERATORS

    ...

    // -- INQUIRIES

    ...
```

```
    // -- OPERATIONS

    ...
}
```

- Don't use standard comments for empty sections.
- Align multiple lines comments with the surrounding statements, start them with an uppercase character and end them with a period.

```
/*
    A long explanation which must be written
    on several lines.
*/

...
```

- Align single line comments with the surrounding statements, start them with an uppercase character and end them with a period.

```
// A short explanation which can be written on a single line.

...
```

- Put end of line comments exactly four spaces after the statement, and start them with lowercase character.

```
DoSomethingWeird();    // a short explanation
```

- Begin C++ header files with `#pragma once`.

```
#pragma once

// -- IMPORTS

#include "tank.hpp"
#include "tank_shell.hpp"
...
```

- Name the unit test class by simply adding a `_TEST` suffix to the class name.

Advices

- Design before you program, to avoid loosing precious time in developing the wrong solution to the wrong problem, by quickly writing :

    - a short text or UI flow explaining how to use the application, to optimize the application interface before implementing it;
    - a short text explaining what the application components will do, to optimize the application architecture before implementing it;
    - a short text or test code explaining how to use the application components, to optimize their external interface before implementing it.
- Develop programs gradually, one feature at a time, using simple and efficient code.
- Don't overgineer your code, choose simple modular designs which can easily be extended.
- Immediately refactor components when their modularity or reusability needs to be improved.
- Instead of adding comments to explain the code intent, refactor the code to make it easy to understand by :
    - choosing better method and parameter names;
    - using local variables to store intermediate results;
    - splitting putting the code of a lengthy method into smaller methods called in sequence.
- Use assertions to check the validity of the method parameters in the debug build, but make the application resilient to external conditions (network failures, missing or corrupted files, etc).

Version

1.0

Author

Eric Pelzer (ecstatic.coder@gmail.com).

License