SFGE C++ Coding Style

Elias Farhan

February 6, 2018

Contents

Introduction

This document is the reference for the coding style, that is a requirement for every commit on `develop` and `master` branches of the Simple and Fun Game Engine (SFGE).

It has been highly inspired by:

- SFML Coding Style
- Unreal Engine Coding Style
- Google C++ Style Guide

SFGE is distributed under the MIT License. Basically, you can do whatever you want as long as you include the original copyright and license notice in any copy of the software/source. You have to include at the beginning of every file of the project this comment:

```
/*
MIT License

Copyright (c) 2017 SAE Institute Switzerland AG

Permission is hereby granted, free of charge, to any
    person obtaining a copy of this software and
    associated documentation files (the "Software"), to
    deal in the Software without restriction, including
    without limitation the rights to use, copy, modify,
    merge, publish, distribute, sublicense, and/or sell
    copies of the Software, and to permit persons to whom
    the Software is furnished to do so, subject to the
    following conditions:

The above copyright notice and this permission notice
    shall be included in all copies or substantial
    portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
     KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
    TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
    PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT
    SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR
    ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
```

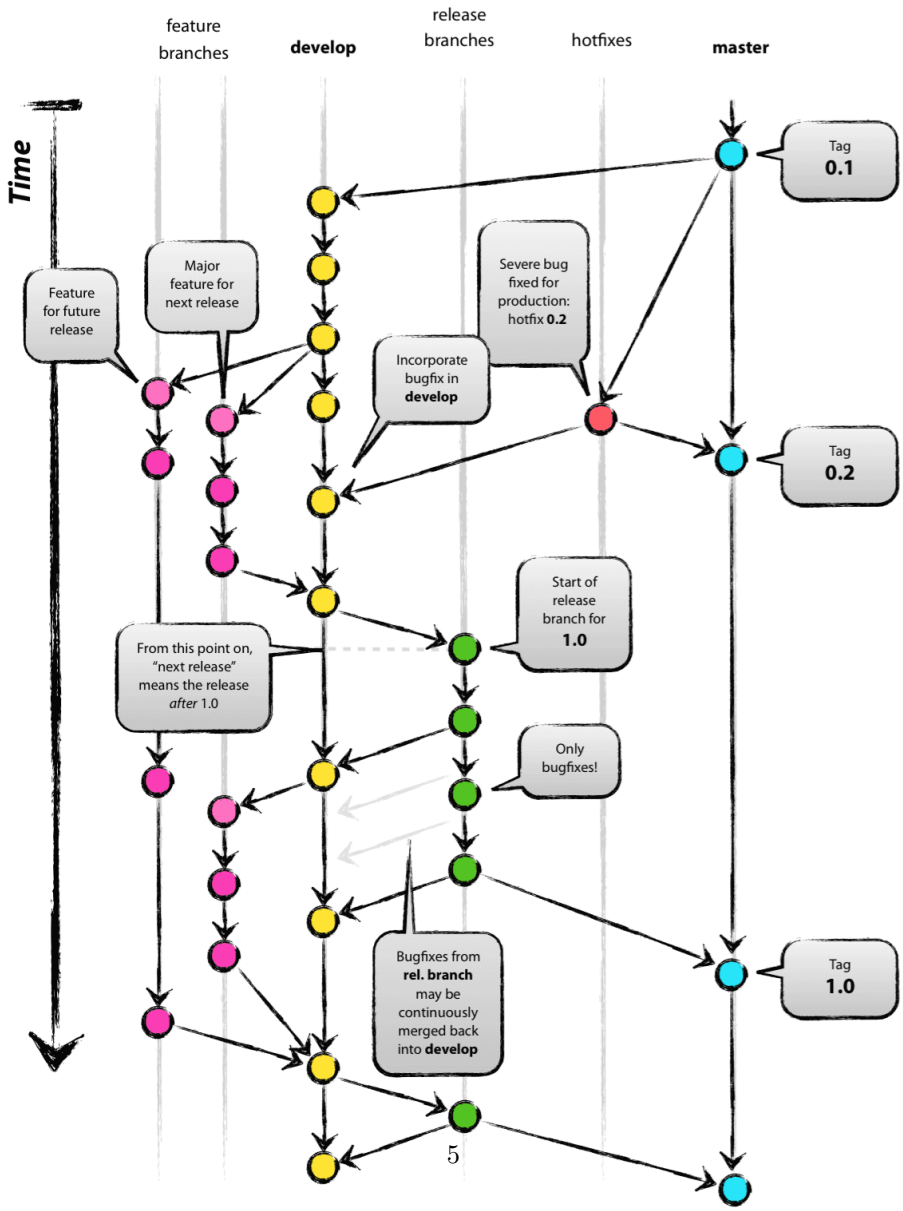
ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

*/

Chapter 1

Git

We are following the git model from Vincent Driessen [1].



1.1 Main branches

At the core, the development model is greatly inspired by existing models out there. The central repo holds two main branches with an infinite lifetime:

- master
- develop

The `master` branch at `origin` should be familiar to every Git user. Parallel to the `master` branch, another branch exists called `develop`.

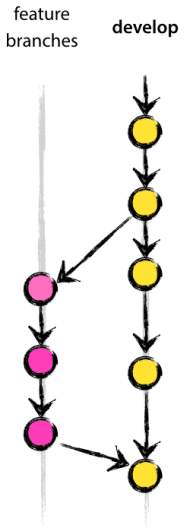
We consider `origin/master` to be the main branch where the source code of `HEAD` always reflects a *production-ready* state.

We consider `origin/develop` to be the main branch where the source code of `HEAD` always reflects a state with the latest delivered development changes for the next release. Some would call this the "integration branch". This is where any automatic nightly builds are built from.

1.2 Supporting branches

The different types of branches we may use are:

- Feature branches
- Release branches
- Hotfix branches



Each of these branches have a specific purpose and are bound to strict rules as to which branches may be their originating branch and which branches must be their merge targets.

1.2.1 Feature branches

May branch off from: `develop`
Must merge back into: `develop`
Branch naming convention: C-style (for example: `prout_foo`) except `master`, `develop`, `release-*`, or `hotfix-*`

Creating a feature branch

When starting work on a new feature, branch off from the `develop` branch.

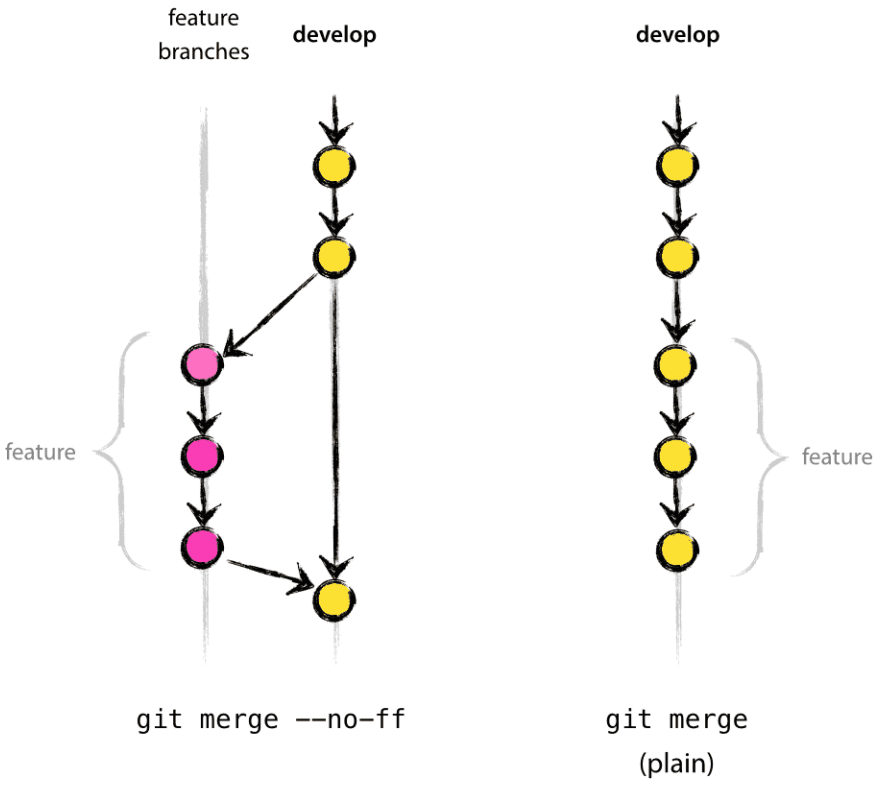
```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

Incorporating a finished feature on develop

Finished features may be merged into the `develop` branch to definitely add them to the upcoming release:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

The `--no-ff` flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature. Compare:



Chapter 2

Syntactical Conventions

2.1 Naming Convention

Type	Convention
file name	prout_foo.cpp, prout_foo.h
type (struct, class, union, enum, typedef)	TitleCase
function, method	TitleCase
local, static and global variable	camelCase
private or protected member	m_TitleCase
enum constant, static const attribute	UPPER_CASE

2.2 Indentation, Space, Parenthesis and Quarks

2.2.1 Tabs vs. Spaces

Tabs should not be used in SFGE code. Indentation is produced by spaces only. A tabulator is equal to 4 spaces. To setup that correctly in Visual Studio 2017: Tools→Options→Text Editor→All Languages→Tabs.

2.2.2 Spaces

The rules are as follows:

- A space precedes an opening parenthesis.
- A space follows a closing parenthesis.
- Rules 1. and 2. are not applied for function calls or declarations.
- A space precedes and follows binary operators and assignment operators.
- A space follows a comma.
- There is no space between a type and its reference & or pointer * specifier.
- A space follows the operator keyword.

- When colon is used for inheritance or with an access modifier it is surrounded by a space on both sides.
- There are no extra spaces at the end of lines.

2.2.3 Declarations

`const` is placed before the type whenever possible. Reference & or pointer * are glued to the type (no extra space).

```
      T          obj;
const T          cobj;
      T&         ref;
const T&         cref;
      T*         ptr;
const T*         cptr;
      T* const   ptrc;
```

When a function or a method is not changing the content of an argument, it must be put as `const`.

```
/**
* \brief Print the values of an vector of int
* \param values The vector of int printed
*/
void PrintValues(const std::vector<int>& values)
{
        for(auto v& : values)
        {
                Log::Msg(v);
        }
}
```

2.2.4 Lines

1. There is only one instruction per line, except for readability in some switches.
2. Every definition (`class`, functions, ...) is followed by an empty line.
3. Braces are placed on new lines by themselves, except for `do ... while` loops.
4. `template` parameters and the rest of the function signature are on two different lines.
5. Every member constructed in the initializer `list` is on a new line.

```
std::list<int> values
{
        1,
        2,
```

```
        3,
        4
};
```

6. If a line is too long it is intelligently broken up into a multi-line statement; e.g.:

```
Color(Uint8(std::min(int(left.r) + right.r, 255)),
      Uint8(std::min(int(left.g) + right.g, 255)),
      Uint8(std::min(int(left.b) + right.b, 255)),
      Uint8(std::min(int(left.a) + right.a, 255)));
```

2.3 Parenthesis and Braces

2.3.1 Blocks

Blocks are always indented by one extra level, except for namespaces when there is only one used in the file.

2.3.2 `if/else/while` Statements

There are two forms of `if/else` statements: single-line or multi-line body. For an `if` statement that has only one instruction no braces are used. In any case a space always separates the keyword from the parenthesis. Every Brace is alone on the line – even if the while body is empty. E.g.:

```
if (audioContext) //Always put the brackets
{
    AlcDestroyContext(audioContext);
}
if (audioContext)
{
    // Set the context as the current one
    // (we'll only need one)
    AlcMakeContextCurrent(audioContext);
}
else
{
    err() << "Failed to create the audio context"
    << std::endl;
}
while ((nanosleep(&ti, &ti) == -1) && (errno == EINTR))
{
}
```

2.3.3 Logical Operators

If multiple && or || operators are used in the same boolean expression, then each part is guarded by parenthesis as soon as they consist of multiple sub-expressions themselves.

```
x < y                 // no parenthesis
(x < y) && (y < z)    // with parenthesis
var && (x < y);       // variable not parenthesized
func() && (x < y);    // function call not parenthesized
```

2.4 Namespaces

The public API lives in the `sfge` namespace. The `sfge::priv` namespace is reserved for implementation details.
Anonymous namespaces are used when global variables are required, or for functions local to the current translation unit, in order to restrict their access to the translation unit.
No `using` directive should be used. Instead the full name is used everywhere.
Like written in subsection 2.3.1, `namespace` blocks are not indented, please change your settings in Visual Studio 2017: Tools→Options→Text Editor→C/C++→Formatting→Indentatio
[ ] Indent namespace contents

```
//Extern dependencies
#include <SFML/Window.hpp>
#include <imgui-SFML.h>
#include <imgui.h>
//Engine dependencies
#include <input/input.h>
#include <engine/log.h>



namespace sfge
{
```

Chapter 3

Documentation

3.1 Doxygen

SFGE uses Doxygen to generate the documentation in HTML or in LaTeX. Each `classe`, `struct`, `enum class`, function, method must be documented with a brief description, the parameters and the returned value for functions and methods.

The comment should be before the definition of the type with a comment starting with two stars and one star per line until the last line with a star and slash.

```
/**
* \brief The Texture Manager is the cache module of all
* the textures used for sprites or other objects
*
*/
class TextureManager : public Module<TextureManager>
{
public:
        /**
        * \brief load the texture from the disk or the
        * texture cache
        * \param filename The filename string of the
        * texture
        * \return The strictly positive texture id > 0,
        * if equals 0 then the texture was not loaded
        */
        unsigned int LoadTexture(std::string filename);
        /**
        * \brief unload the texture by removing a
        * reference count, if reference count is 0 then
            it
        * is unloaded from the cache
        * \param text_id The texture id striclty positive
        *
        */
```

```
        void UnloadTexture(unsigned int text_id);
        /**
        * \brief Used after loading the texture in the
        * texture cache to get the pointer to the texture
        * \param text_id The texture id striclty positive
        * \return The pointer to the texture in memory
        */
    sf::Texture* GetTexture(unsigned int text_id);
private:

    std::map<std::string, unsigned int> nameIdsMap;
    std::map<unsigned int, sf::Texture> texturesMap;
        std::map<unsigned int, unsigned int> refCountMap;
        unsigned int increment_id = 0;
};
```

3.2 Guidelines

- Write self-documenting code:

```
// Bad:
t = s + l - b;

// Good:
totalLeaves = smallLeaves + largeLeaves
- smallAndLargeLeaves;
```

- Write useful comments:

```
// Bad:
// increment Leaves
++leaves;

// Good:
// we know there is another tea leaf
++leaves;
```

- Do not comment bad code - rewrite it:

```
// Bad:
// total number of leaves is sum of
// small and large leaves less the
// number of leaves that are both
t = s + l - b;

// Good:
totalLeaves = smallLeaves + largeLeaves
- smallAndLargeLeaves;
```

- Do not contradict the code:

```
// Bad:
// never increment Leaves!
++leaves;

// Good:
// we know there is another tea leaf
++leaves;
```

Chapter 4

Data Structures

4.1 Structures and Classes

`structs` are used to wrap up one or more variables together but do not use encapsulation; they are generally used by `classes` that do protect their members with protected or private modifiers. `structs` can not have constructors and should not have methods. They do not use access specifiers or inheritance.
In a `class`, the public interface comes first (usually with constructors at the top), followed by protected members and then private data. In a given access-modifier group static members are grouped together.

Chapter 5

Header

5.1 Includes

The inclusion order is as follows:

1. Externals headers
2. Standard library headers
3. SFGE headers

Example:

```
//Externals includes
#include <SFML/Graphics.hpp>
//STL includes
#include <list>
//SFGE includes
#include <engine/log.h>
```

5.2 Class Definitions

In a class, the public interface comes first (usually with constructors and special member functions at the top), followed by protected members and then private data. In a given access-modifier group, static members are grouped together.

```
////////////////////////////////////////////////////////////
//
// License text...
//
////////////////////////////////////////////////////////////

#ifndef SFGE_FILENAME_H
#define SFGE_FILENAME_H

//Externals includes
#include <...>
```

```
//STL includes
#include <...>
//SFGE includes
#include <...>
```

Bibliography

[1] **A successful Git branching model**, *Vincent Driessen*, Accessed [03.10.2017]. Link:
`http://nvie.com/posts/a-successful-git-branching-model/`

[2] **SFML Code Style Guide**, *Laurent Gomilla*, Accessed [03.10.2017]. Link:
`https://www.sfml-dev.org/style.php`

[3] **Unreal Coding Standard**, Epic Games, Accessed [03.10.2017]. Link:
`https://docs.unrealengine.com/latest/INT/Programming/Development/CodingStandard/`

[4] **Google C++ Style Guide**, Google, Accessed [03.10.2017]. Link:
`https://google.github.io/styleguide/cppguide.html`